**📘 Java Functional Programming, Lambda Expressions, and Stream API**

**Functional Programing**

**What is Functional Programming?**

Functional programming (FP) is a programming paradigm in which:

1. **Functions are “first-class citizens.”** You can pass functions as arguments, return them from other functions, and store them in data structures.
2. **Emphasis on “pure” functions.** A pure function’s output depends solely on its input and has no side effects—meaning it doesn’t modify external state.
3. **Immutable data.** Variables (or data structures) are generally not modified after they’re created; instead, you create new instances with updated values.
4. **Declarative style.** You focus on *what* needs to be done, rather than detailing *how* to do it step by step.

**Functional Programming in Java**

Although Java is predominantly an OOP language, since Java 8 it has introduced several features to support a functional style:

* **Lambda Expressions**: Compact syntax to create instances of functional interfaces (e.g., Predicate<T>, Function<T,R>).
* **Method References**: A shorthand for referencing existing methods as lambdas.
* **Stream API**: Enables a functional, pipeline-based approach to manipulate and process data collections (filter, map, reduce).
* **Functional Interfaces**: Interfaces with a single abstract method (SAM). Common examples include Predicate<T>, Function<T,R>, Consumer<T>, and Supplier<T>.

**When to Use Which Paradigm?**

* **Traditional OOP** fits well where your domain naturally models real-world objects with varying states—like building complex enterprise systems with well-defined entities.
* **Functional Programming** excels in scenarios involving heavy data transformations, concurrency, or parallelization. By leveraging pure functions and immutability, you reduce the risk of hidden side effects that can lead to bugs.

Modern Java applications often blend both paradigms:

* Use OOP constructs (classes and objects) to model your domain.
* Use functional concepts (lambdas, streams, immutable data structures) for data processing tasks or whenever side-effect-free operations make code simpler and safer.

**Summary**

Functional programming in Java brings a new perspective to writing cleaner, more predictable, and often more concise code. While Java remains fundamentally object-oriented, its functional features (lambda expressions, Stream API, and functional interfaces) enable developers to mix FP with OOP. Understanding the differences—especially around immutability, side effects, and the declarative style—can help you choose the most appropriate approach for each part of your application.

**📘 Java Functional Programming, Lambda Expressions, and Stream API**

This document provides **comprehensive notes** with a **step-by-step learning path** for understanding **Java Functional Programming**, **Lambda Expressions**, and **Stream API**. The notes include **pre-requisite concepts**, detailed explanations, and **examples for each topic**.

**🧩 Learning Path and Priority**

**✅ Step 1: Lambda Expressions**

* Pre-requisites: **Anonymous Classes**, **Functional Interfaces**
* Core concept for functional programming in Java.

**✅ Step 2: Functional Programming**

* Builds on **Lambda Expressions** to introduce the functional programming paradigm.

**✅ Step 3: Stream API**

* Uses **functional programming concepts** to process collections of data efficiently.

**📘 Step 1: Understanding Lambda Expressions**

Before diving into **Lambda Expressions**, let’s first understand **Anonymous Classes** and **Functional Interfaces**.

**Pre-Requisite 1: Anonymous Classes**

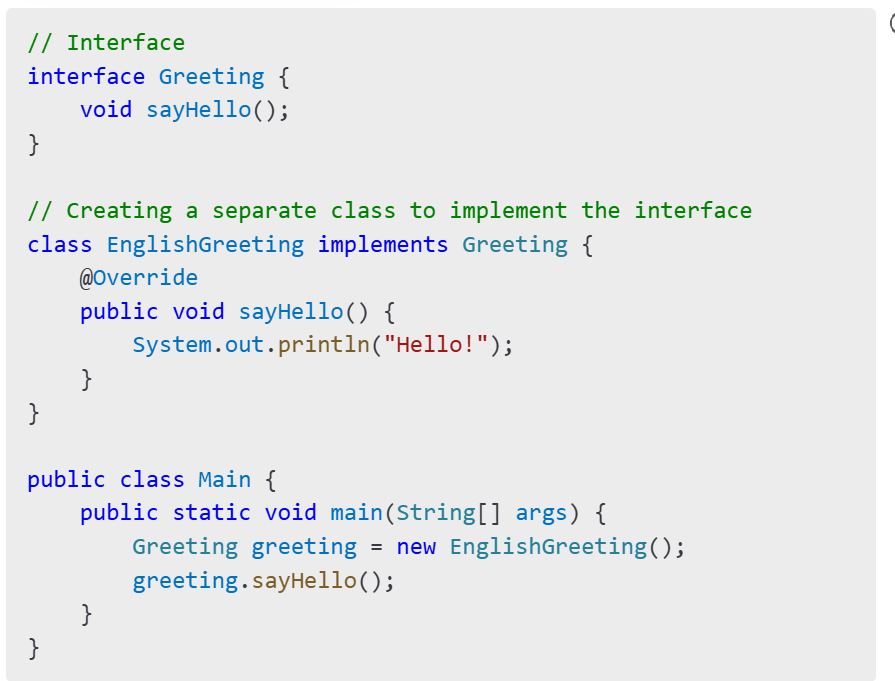
**🔧 What is an Anonymous Class?**

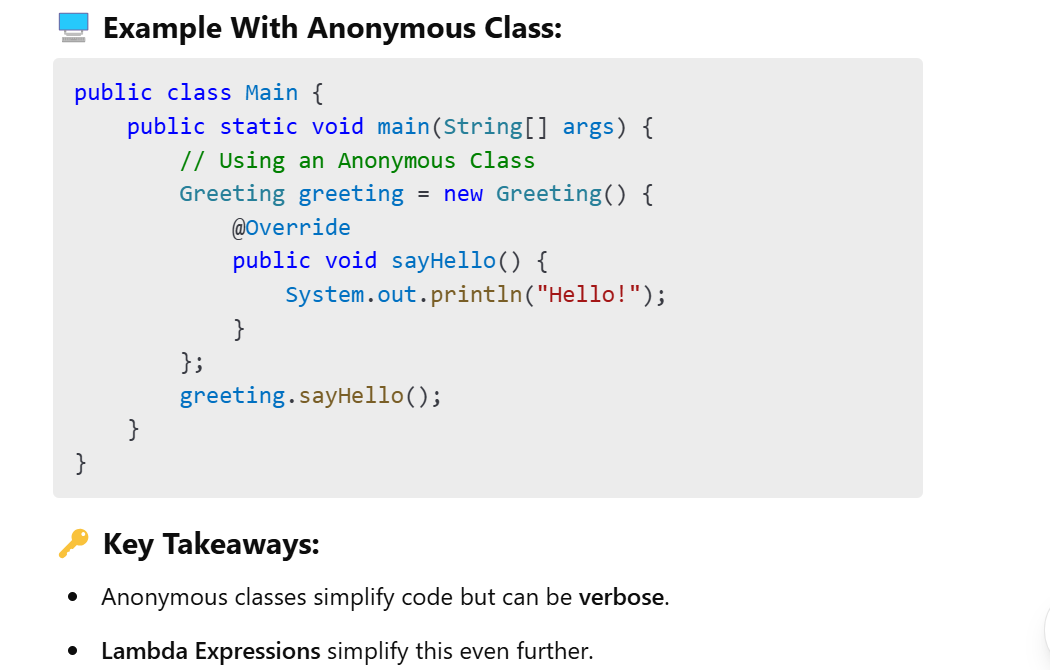
An **Anonymous Class** is a class without a name. It is often used to create an **implementation of an interface or abstract class** on the fly.

**✅ Why Do We Need Anonymous Classes?**

* To create **quick implementations** of interfaces or abstract classes.
* To avoid creating **separate class files** for simple tasks.

**🖥️ Example Without Anonymous Class:**







It **looks like you're creating an object of the Greeting interface**, but you’re actually creating an instance of a **class that implements the interface** on the fly. Let me clarify this step by step.

**🔍 How is it possible to create an object of an interface?**

By itself, **an interface cannot be instantiated**.  
However, **you can create an object of an anonymous class that implements the interface**.  
In this case, you're creating a **nameless class that implements the Greeting interface** and providing an implementation for the sayHello() method.





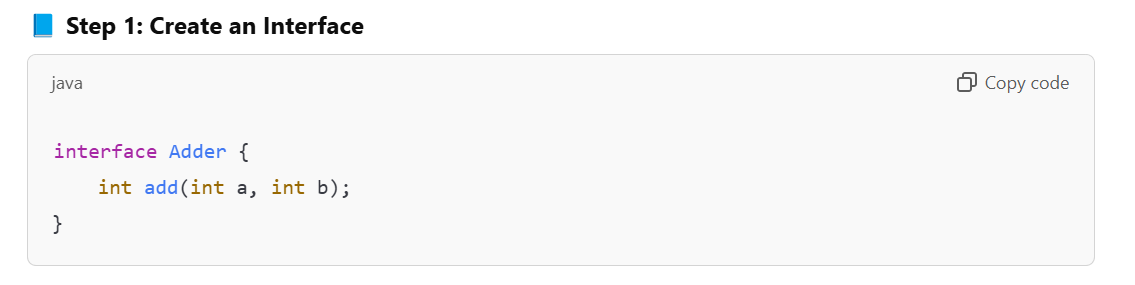
**✅ Why Do We Use Anonymous Classes?**

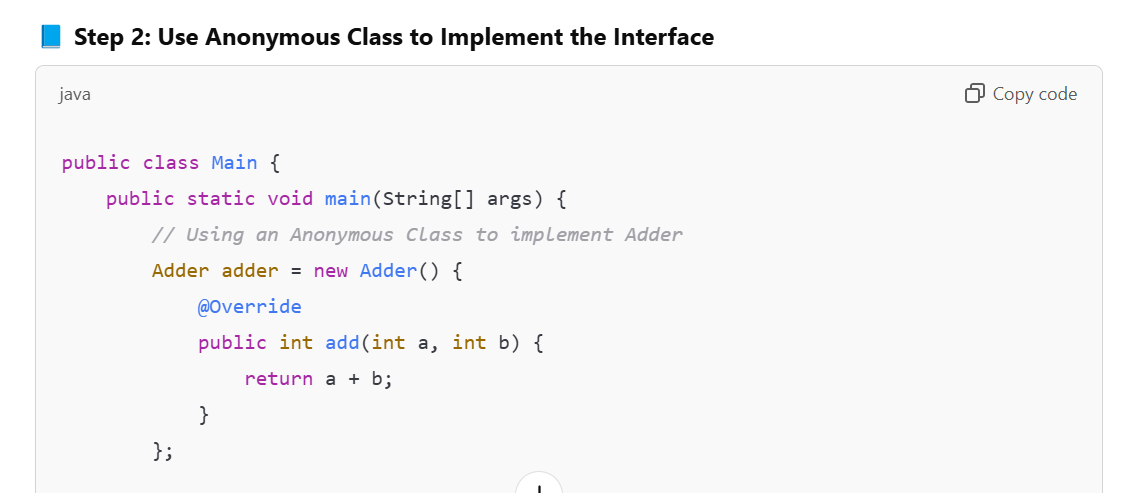
1. **Quick Implementation:**  
   You don't need to create a separate class file to implement a simple interface.
2. **One-Time Use:**  
   It's useful when you need a **one-time implementation** of an interface or abstract class.

**🔎 Summary:**

* **Anonymous Class:** A **nameless class** created to implement an interface or extend a class.
* **Why It Looks Like We’re Creating an Object of an Interface:**  
  We’re actually creating an object of an **anonymous class** that implements the interface.

✅ **Example: Adding Two Integers Using Anonymous Class**

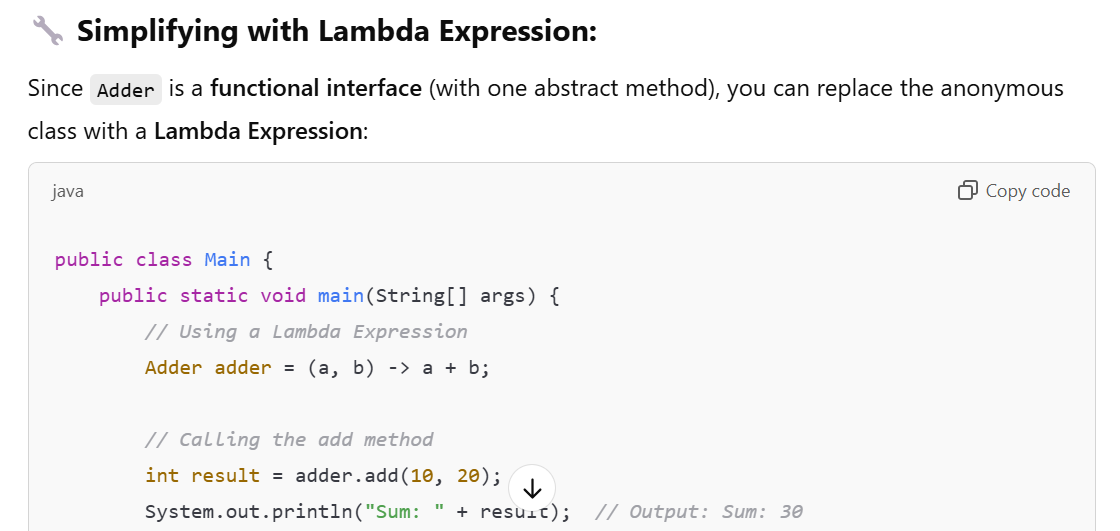






**✅ Breaking It Down:**

1. **Interface Adder:**
   * It has a single abstract method add(int a, int b).
2. **Anonymous Class:**
   * Instead of creating a separate class to implement Adder, we create an **anonymous class** that provides an **implementation for the add() method** directly inside the main() method.
3. **Calling the Method:**
   * We call adder.add(10, 20) to get the sum of two numbers.



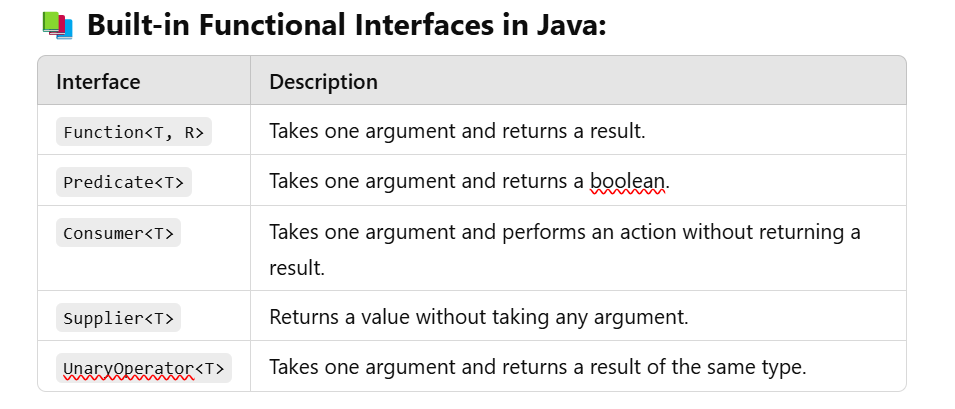
**🔎 Pre-Requisite 2: Functional Interfaces**

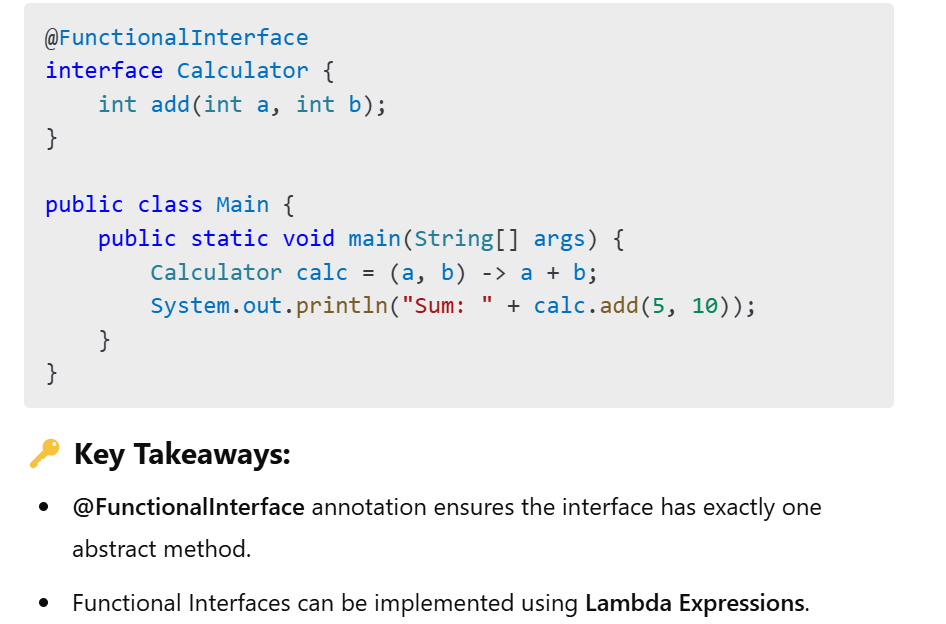
**🔧 What is a Functional Interface?**

A **Functional Interface** is an interface that has **exactly one abstract method**. It can have **default** and **static methods**, but only **one abstract method**.

**✅ Why Functional Interfaces?**

* Functional Interfaces are the foundation of **Lambda Expressions**.
* They allow you to pass **functions as arguments** in Java.





**📘 Java Lambda Expressions**

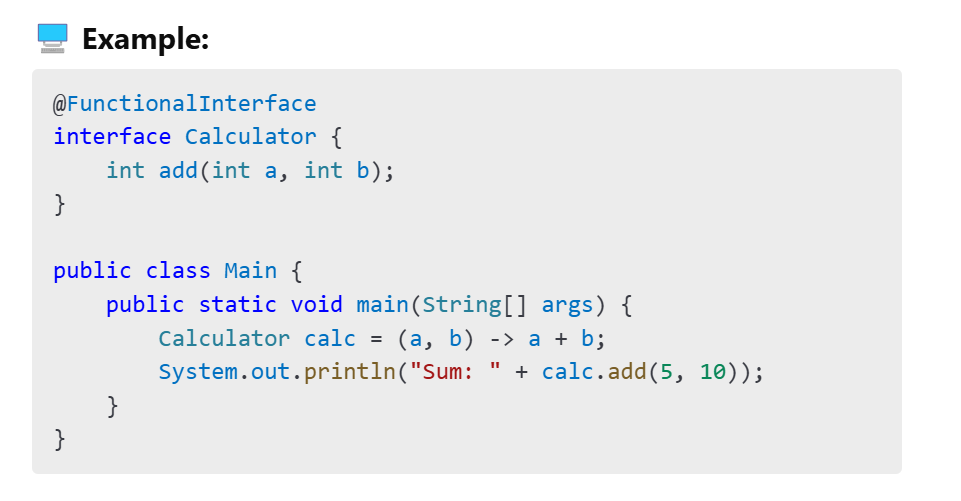
**🔎 What is a Lambda Expression?**

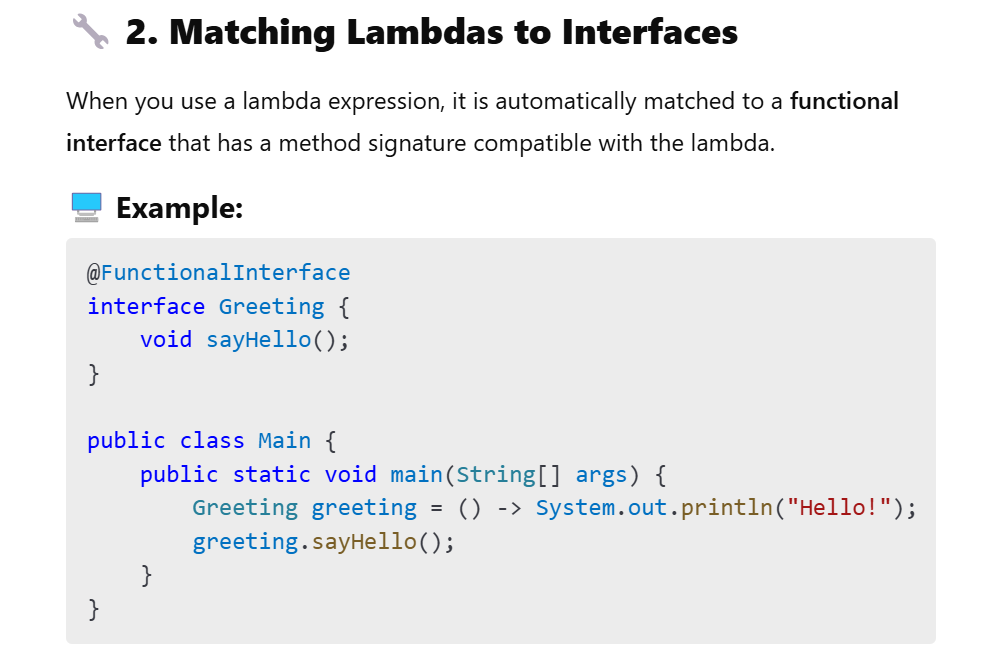
A **Lambda Expression** is a way to represent an **anonymous function** in Java. It allows you to pass **functions as arguments** to methods and write more concise, readable code.

**🔧 1. Java Lambdas and the Single Method Interface**

A **Functional Interface** is an interface that has **exactly one abstract method**. Lambda expressions can be used to implement functional interfaces.

**🖥️ Example:**



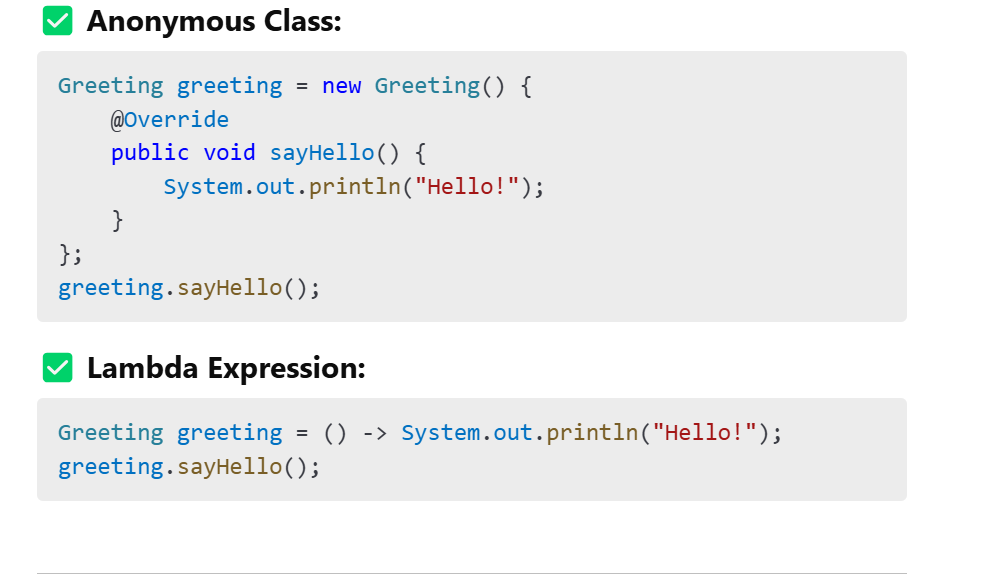


**🔧 3. Interfaces with Default and Static Methods**

Functional Interfaces can have **default** and **static methods**. These methods do not affect the lambda compatibility because they are not abstract methods.

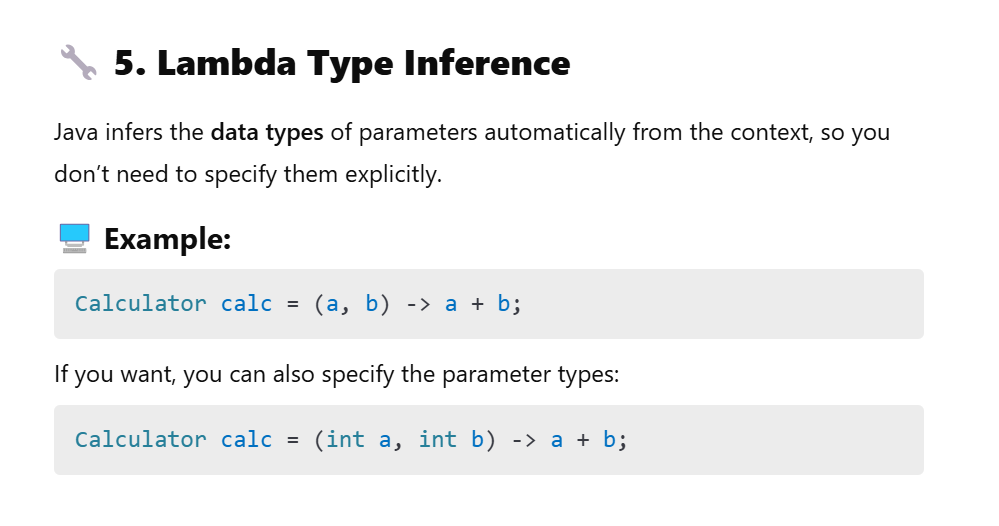


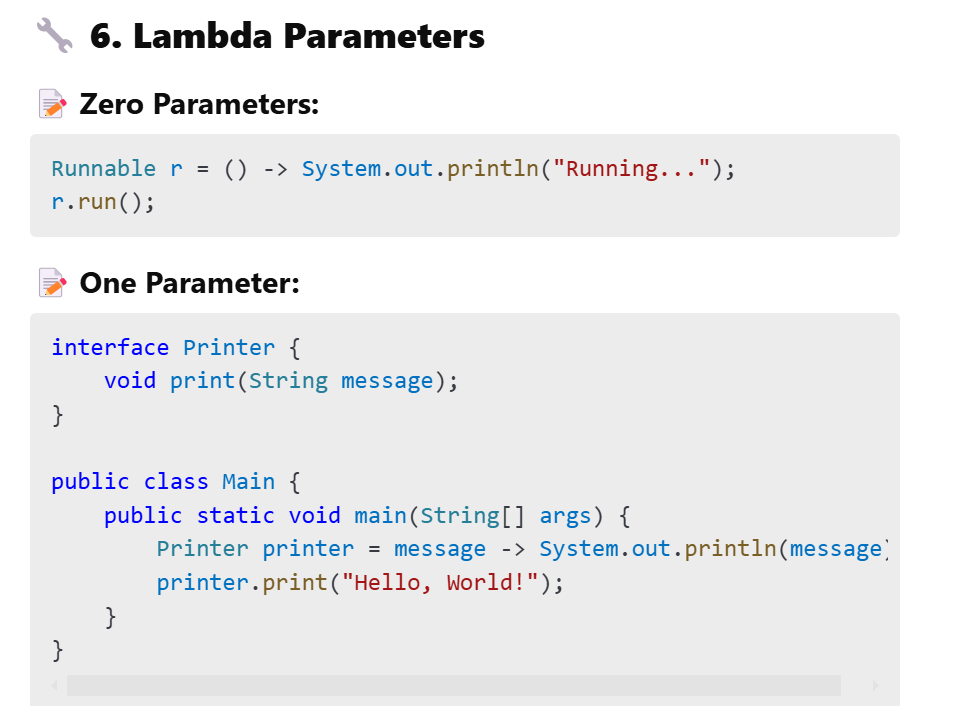
**🔧 4. Lambda Expressions vs. Anonymous Interface Implementations**

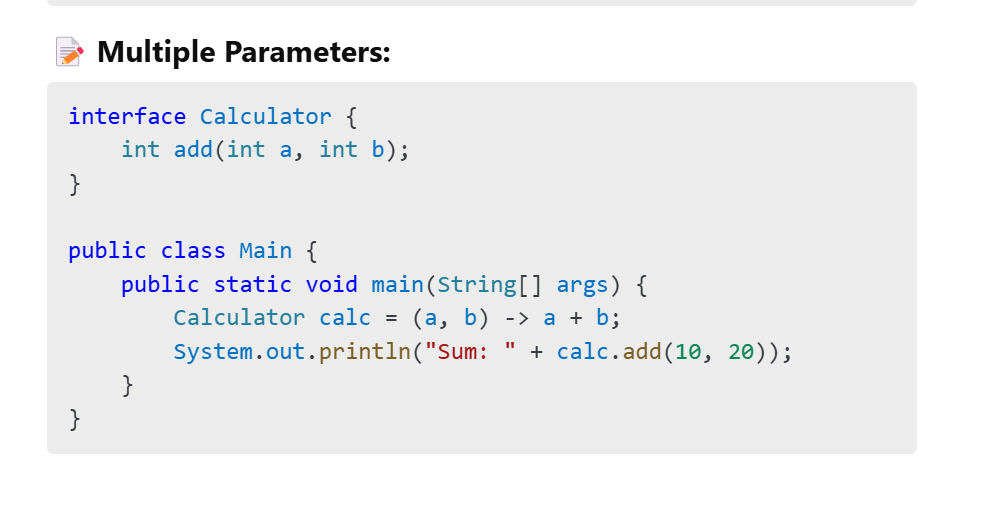


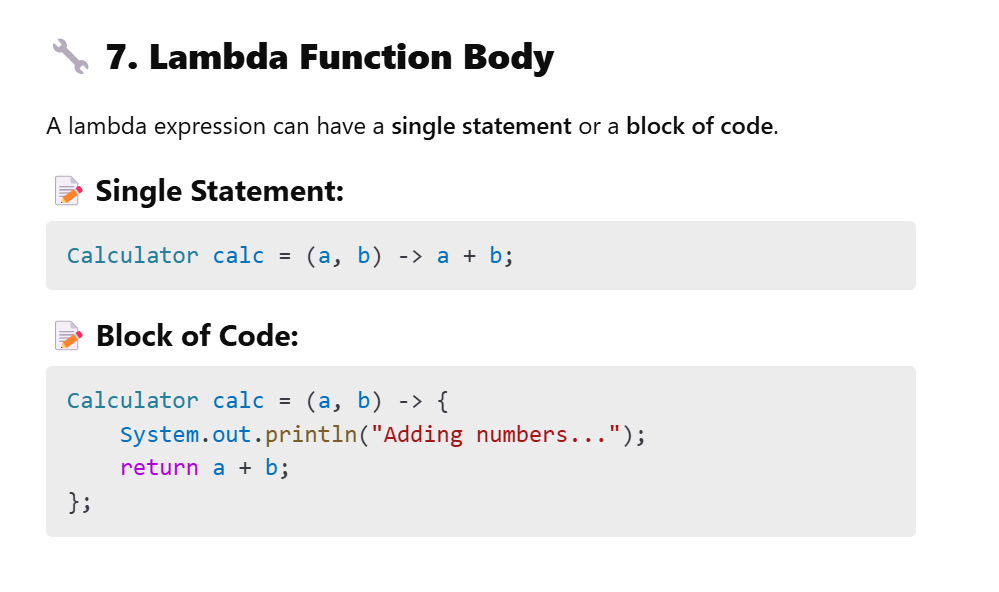
**🔧 5. Lambda Type Inference**

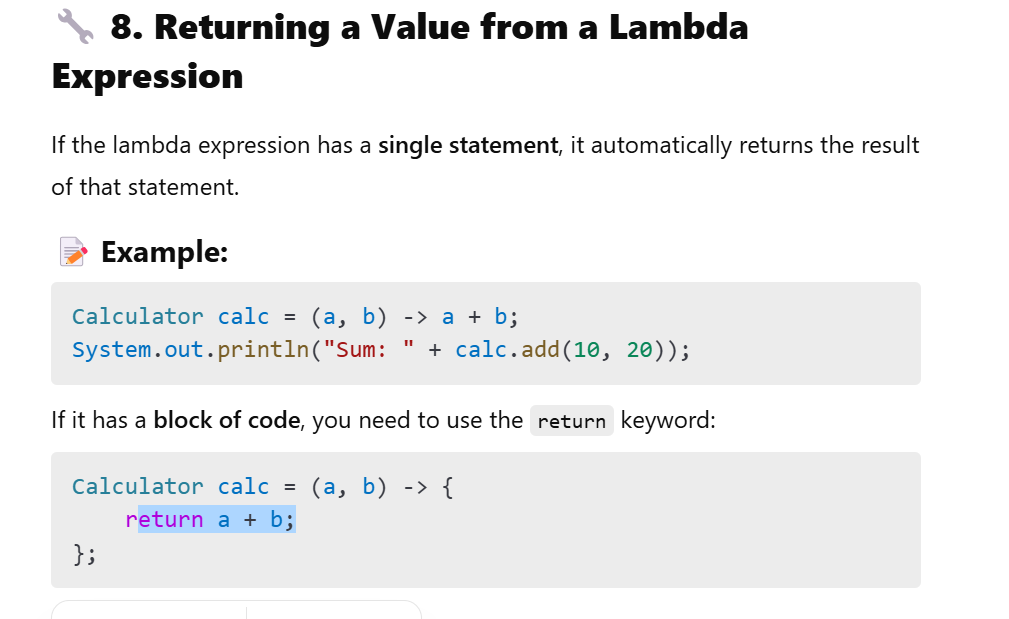
Java infers the **data types** of parameters automatically from the context, so you don’t need to specify them explicitly.











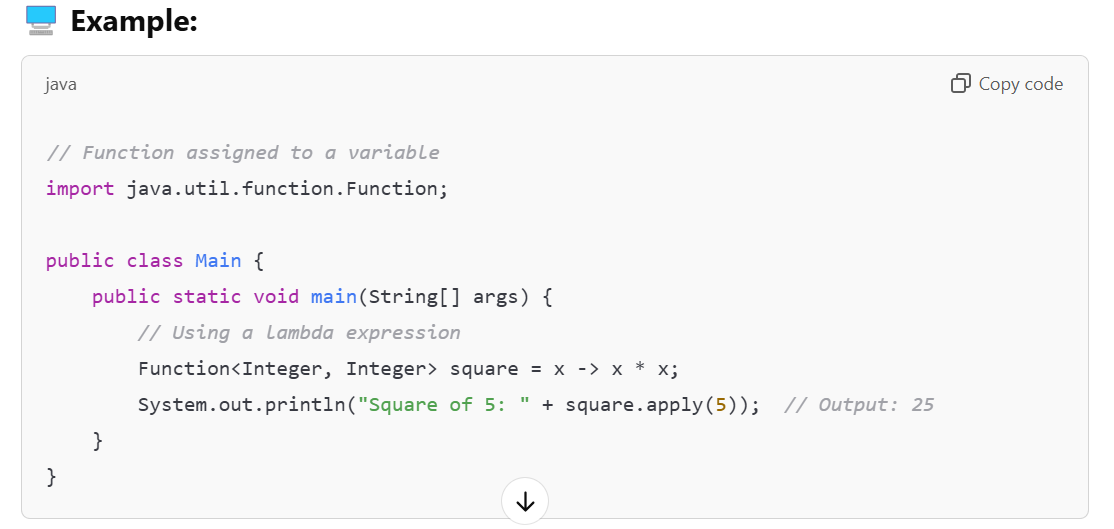
Let's go through **Functional Programming Concepts** with detailed explanations and examples.

**📘 1. Functional Programming Basics**

**Functional Programming (FP)** is a programming paradigm where you treat **functions as first-class citizens**. This means functions can be:

* **Assigned to variables**
* **Passed as arguments**
* **Returned from other functions**

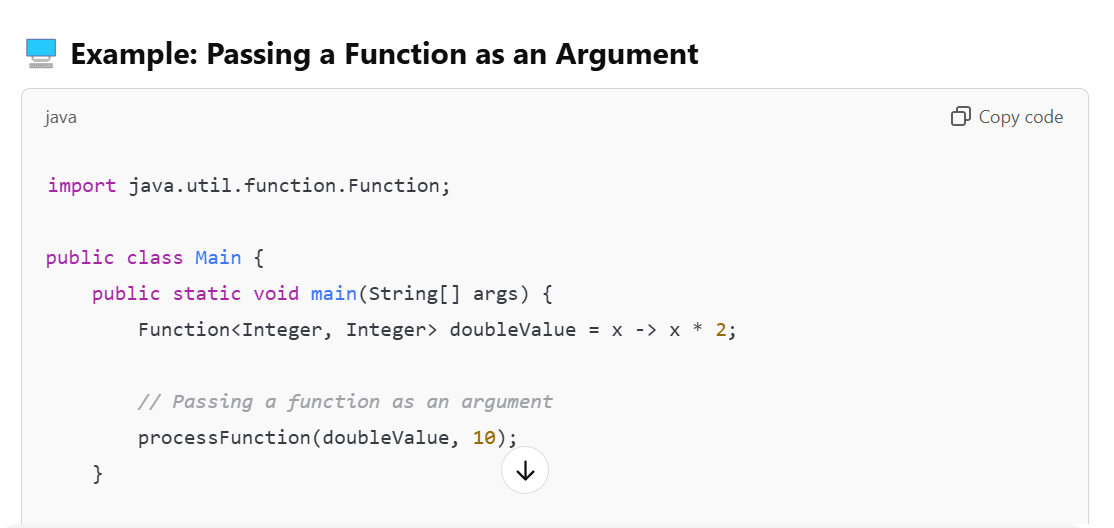
FP focuses on **writing clean, predictable, and side-effect-free code** by using **pure functions** and **immutable data**.

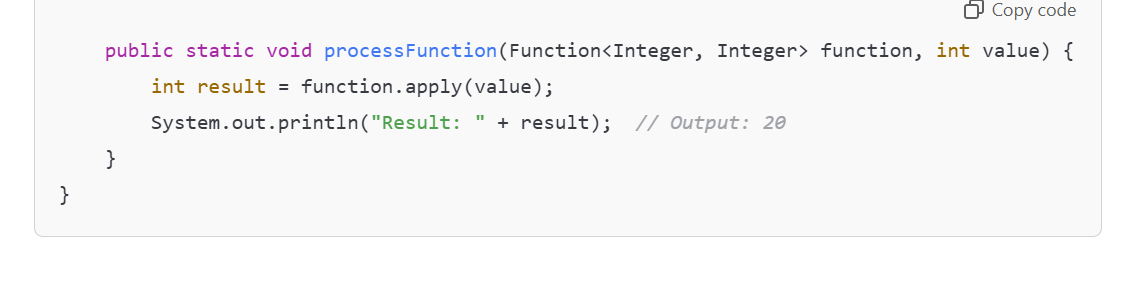


**📘 2. Functions as First-Class Objects**

In Java, **functions are treated as first-class objects**. This means:

* Functions can be **assigned to variables**.
* Functions can be **passed as arguments** to other functions.
* Functions can be **returned from other functions**.

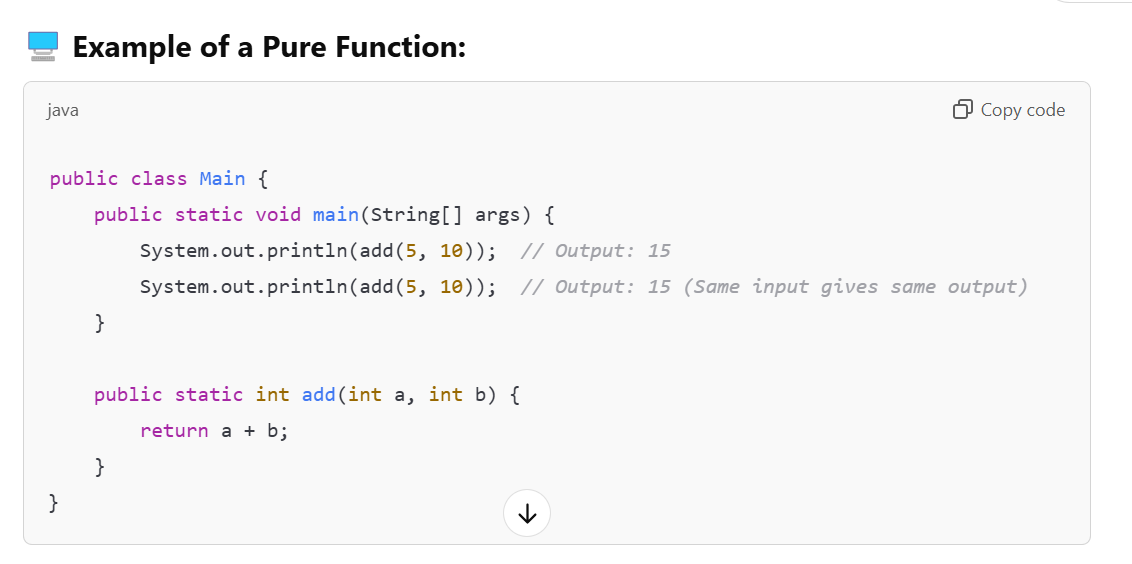


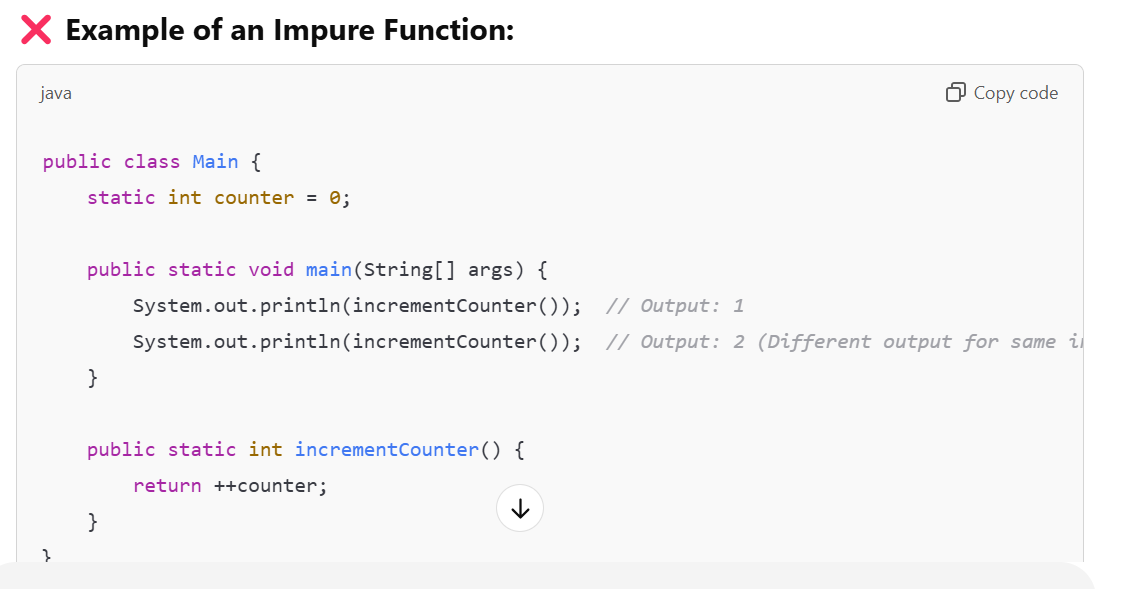


**📘 3. Pure Functions**

A **pure function** is a function that:

* **Always returns the same result** for the same inputs.
* **Has no side effects** (does not modify external state).

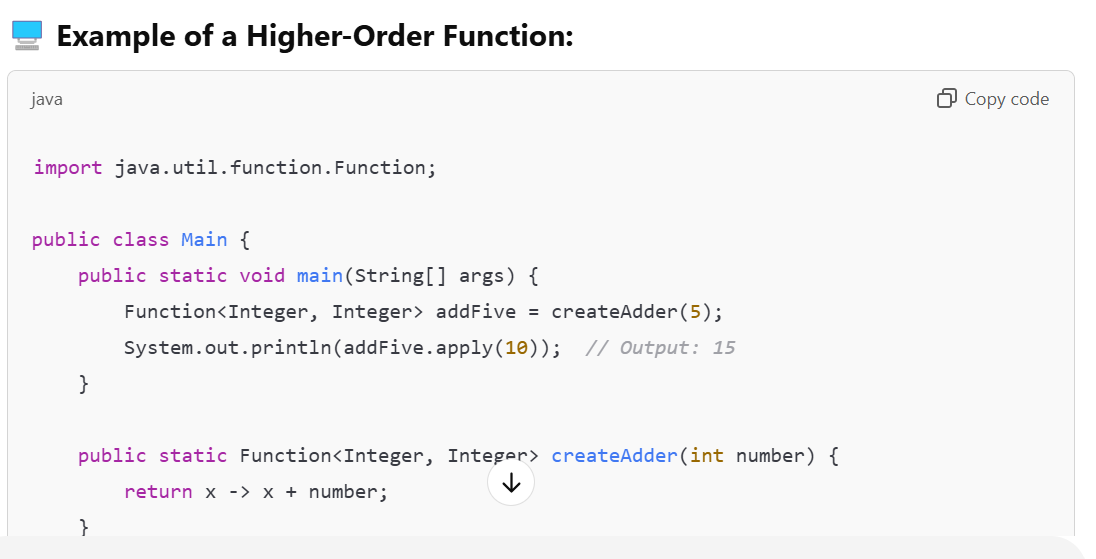




**📘 4. Higher-Order Functions**

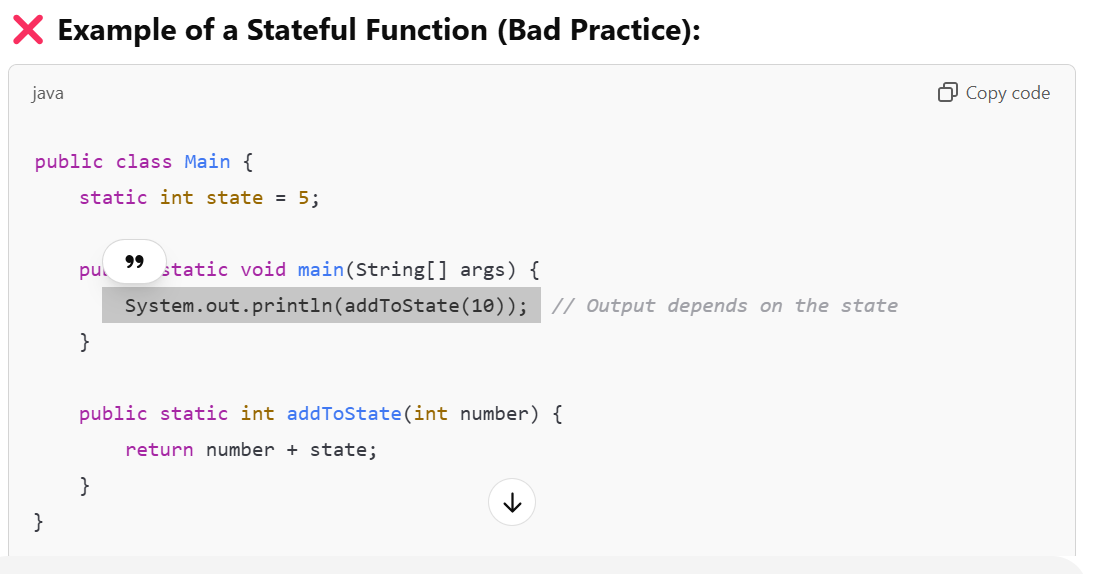
A **Higher-Order Function** is a function that:

* **Takes another function as an argument**.
* **Returns another function as a result**.



**📘 5. No State**

In functional programming, functions should be **stateless**. This means they should not rely on any external state that can change.

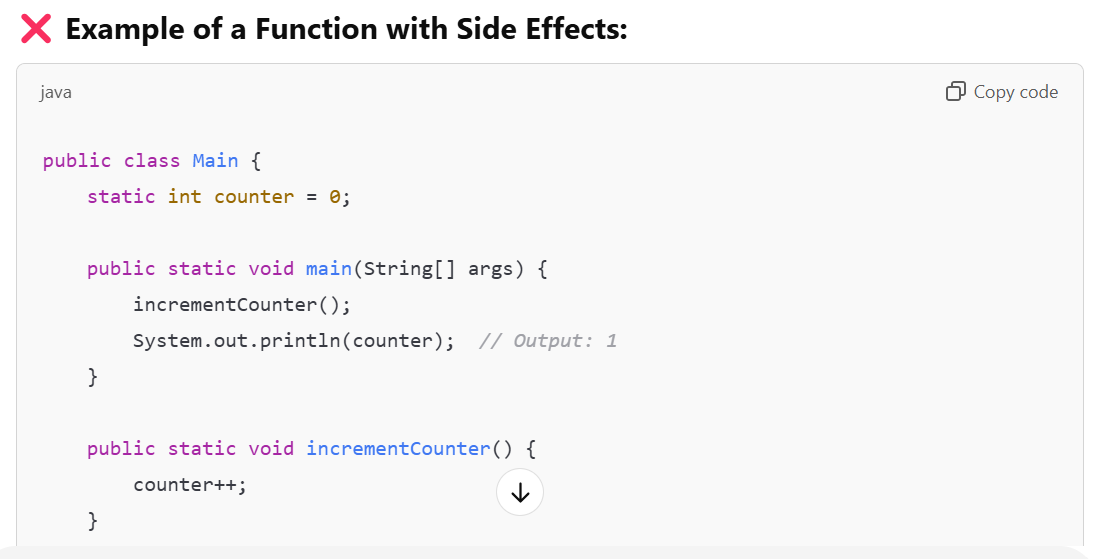




**📘 6. No Side Effects**

A **side effect** occurs when a function modifies an external state or interacts with the outside world (e.g., changing a global variable or writing to a file).

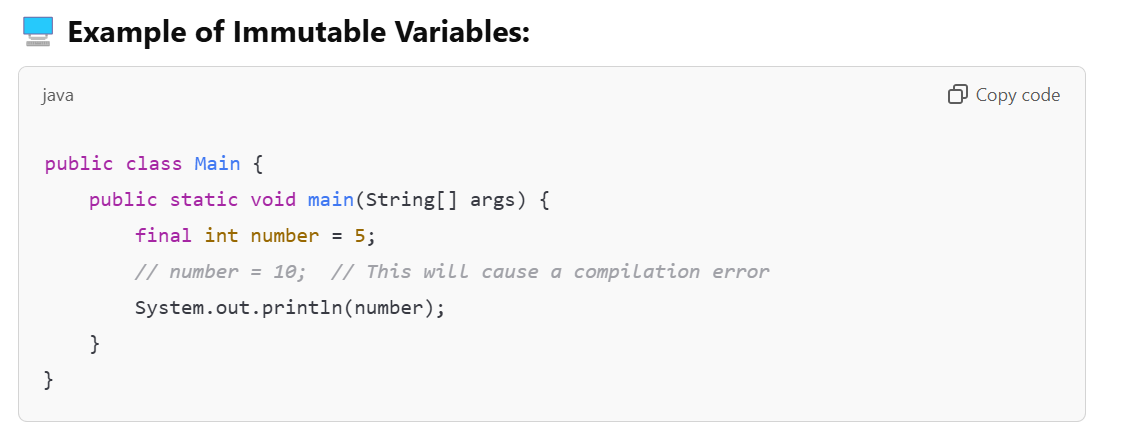
Functional programming aims to write functions with **no side effects**.





**📘 7. Immutable Variables**

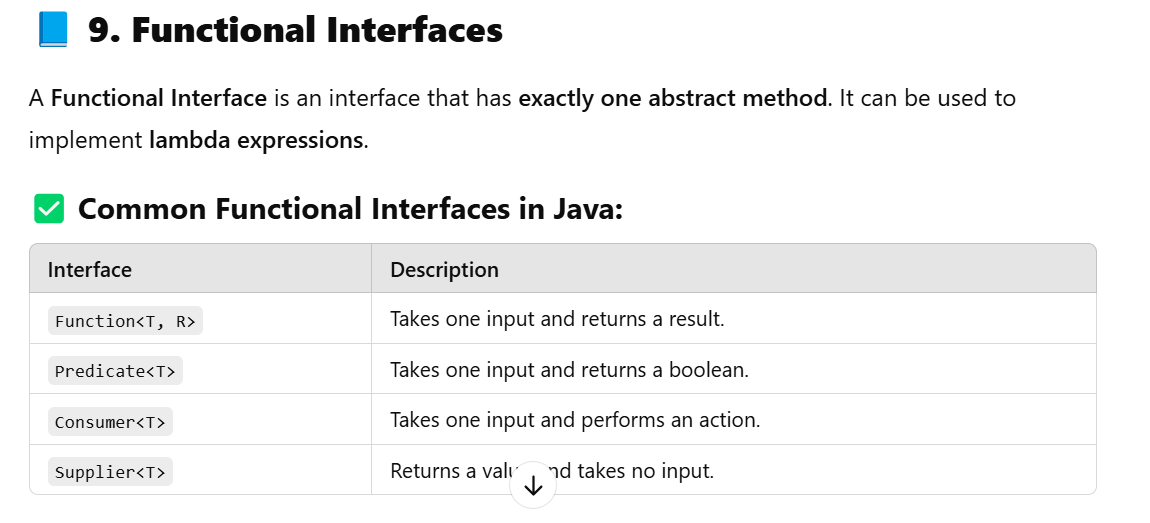
In functional programming, **immutability** means that once a variable is created, its value cannot be changed.

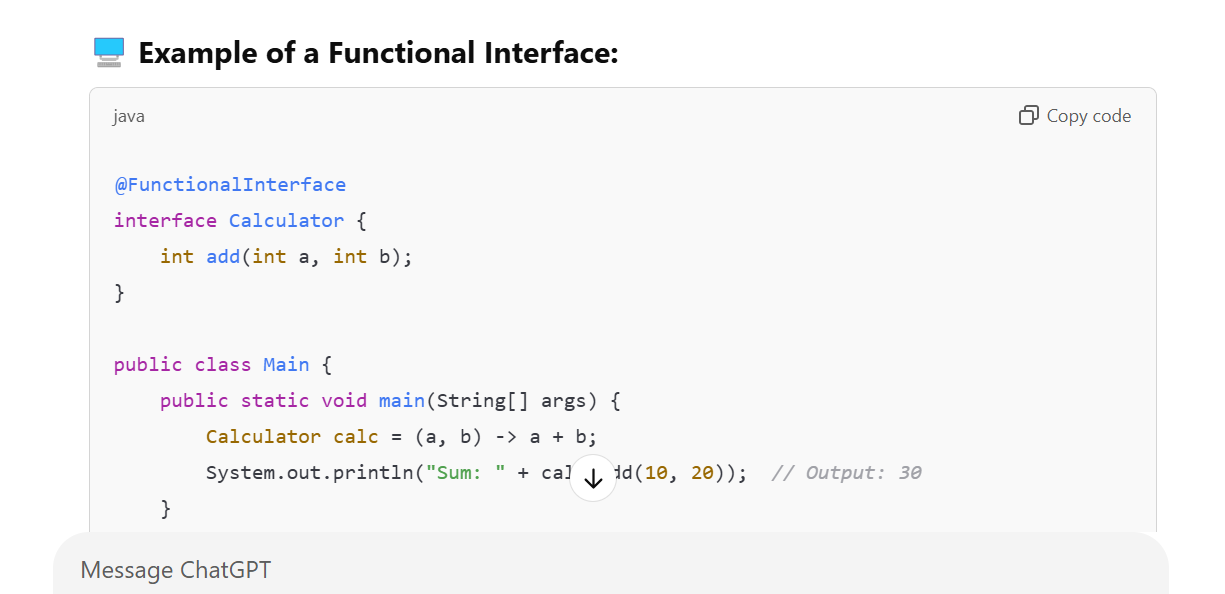


**📘 8. Favour Recursion Over Looping**

Functional programming favours **recursion** instead of **loops** for repetitive tasks.







**📘 Java Stream API Tutorial**

The **Java Stream API** is introduced in **Java 8** to processs collections of data in a **declarative way**. It allows you to perform **filtering**, **mapping**, **reducing**, and other operations on collections like **Lists**, **Sets**, and **Maps**.

**✅ Java Stream Definition**

A **Stream** is a sequence of **elements** that can be processed in parallel or sequentially. It **does not store data** but provides a way to perform computations on data in a functional programming style.



**✅ Stream Processing**

Stream processing involves **creating**, **modifying**, and **consuming** streams using various **operations**.

There are two types of stream operations:

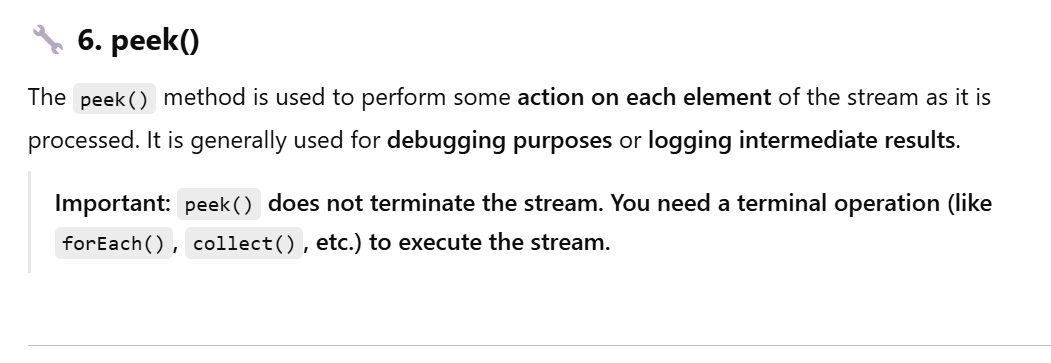
1. **Non-Terminal Operations** (Intermediate Operations)
   * Return a new stream and are **lazy**.
2. **Terminal Operations**
   * Return a **result** (like a list, count, or boolean) and **consume the stream**.

**✅ How to Obtain a Stream**

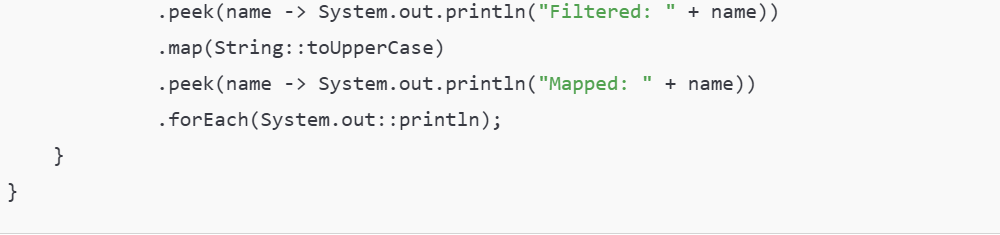
You can obtain a stream in various ways:

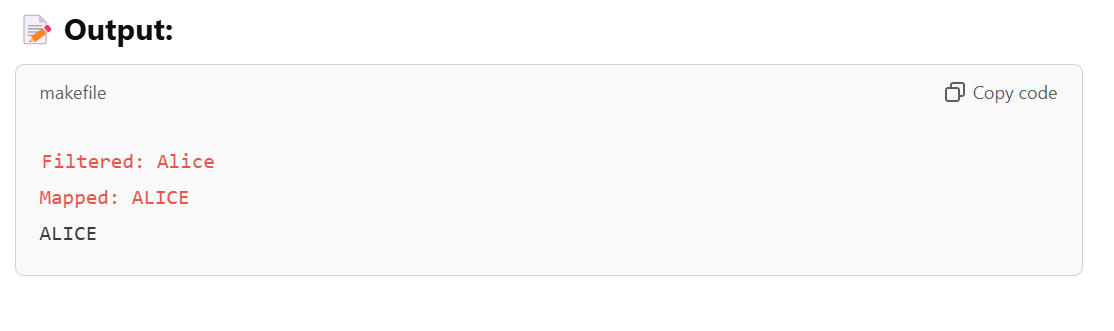
1. **From a Collection**
2. **From Arrays**
3. **From Values**
4. **From Files**
5. **From a Builder**

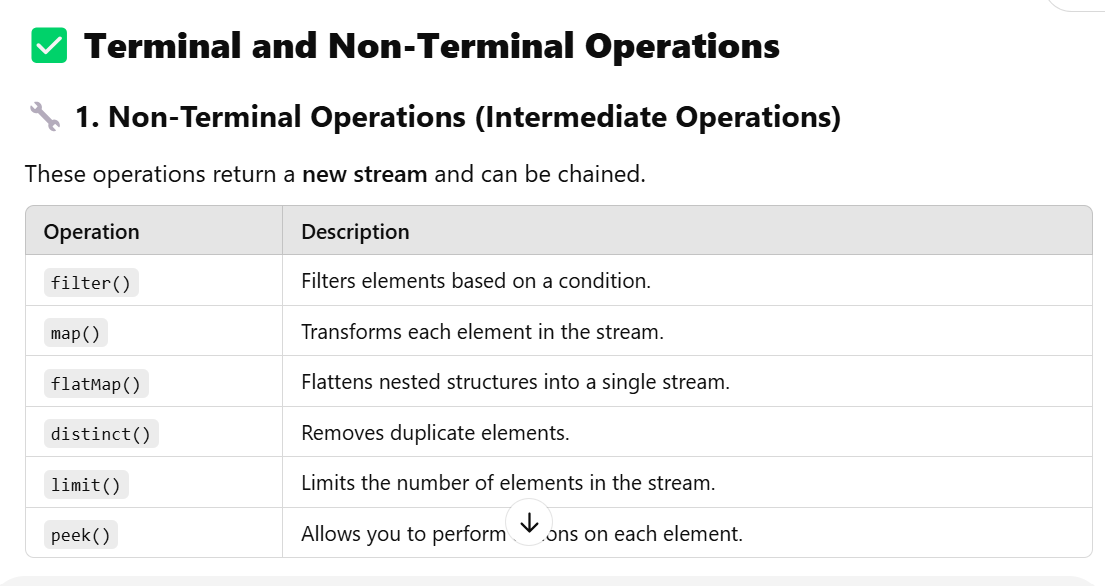


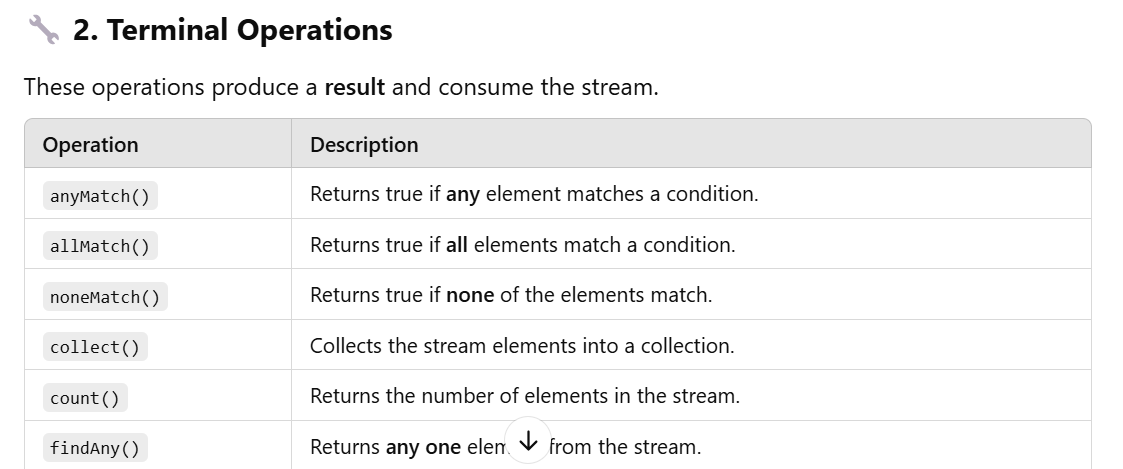


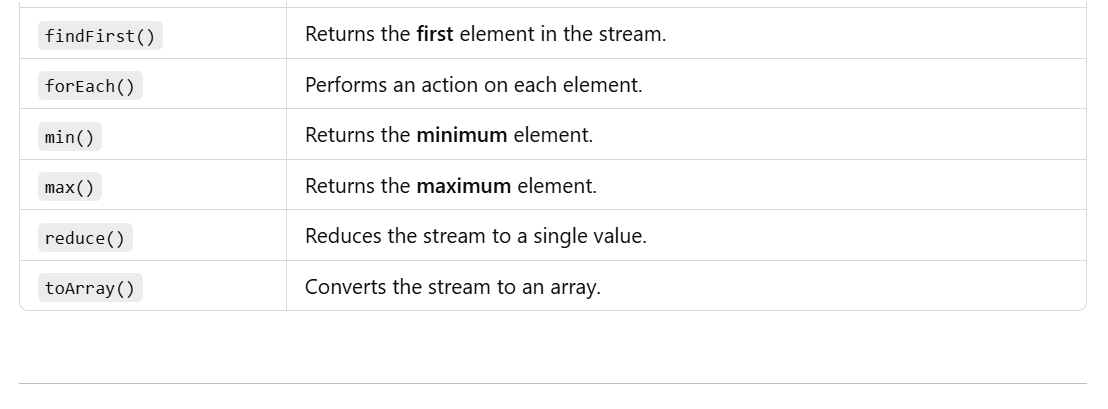








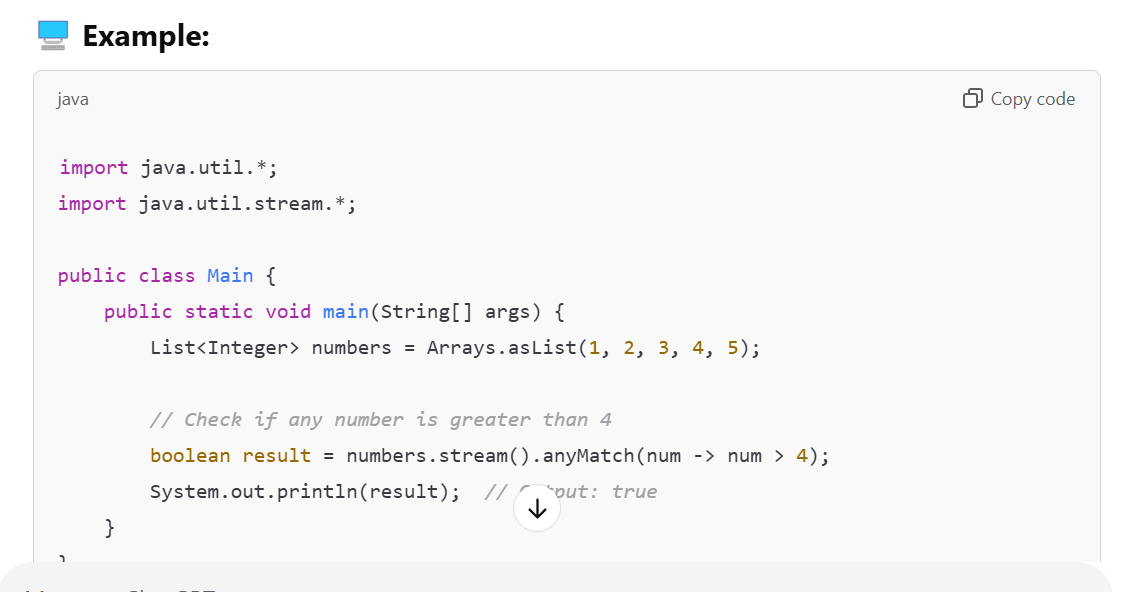


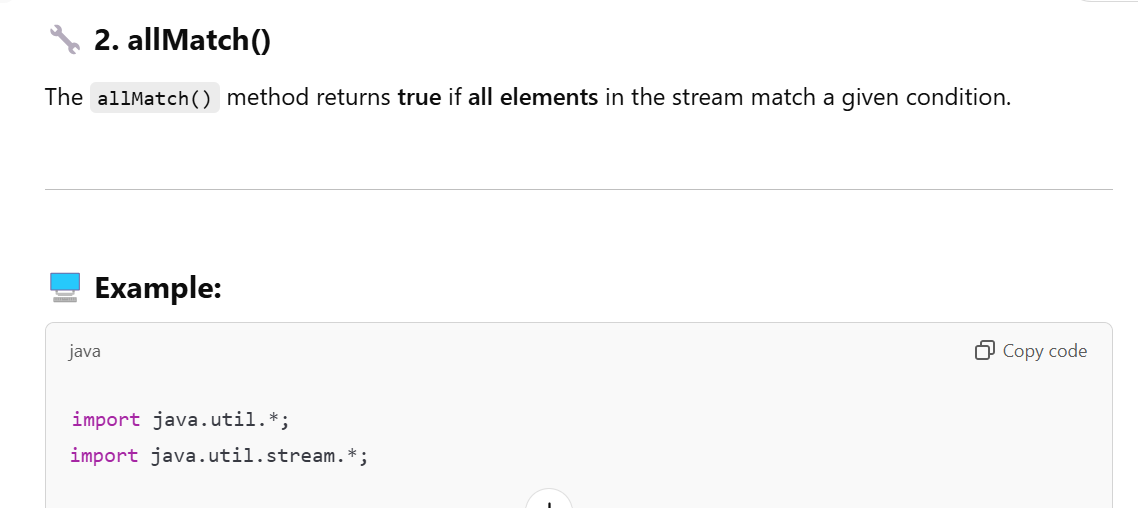


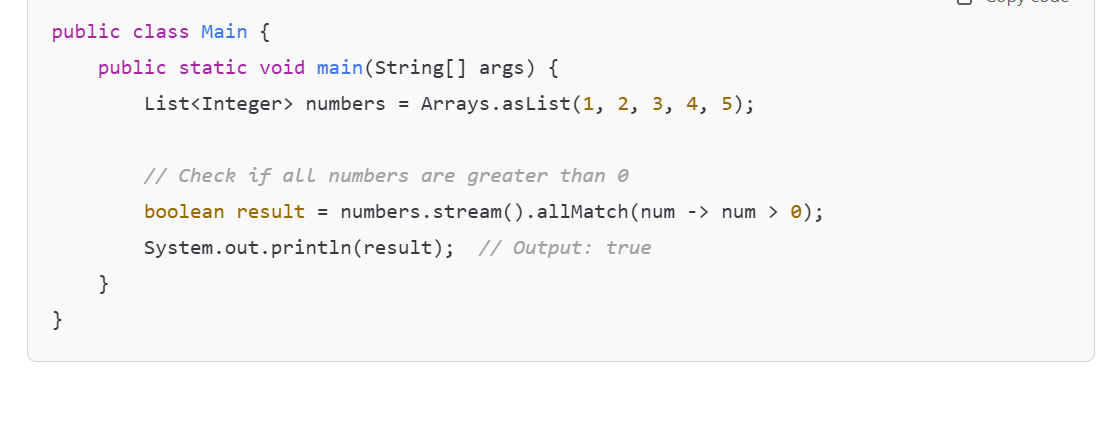
**✅ Terminal Operations with Examples**

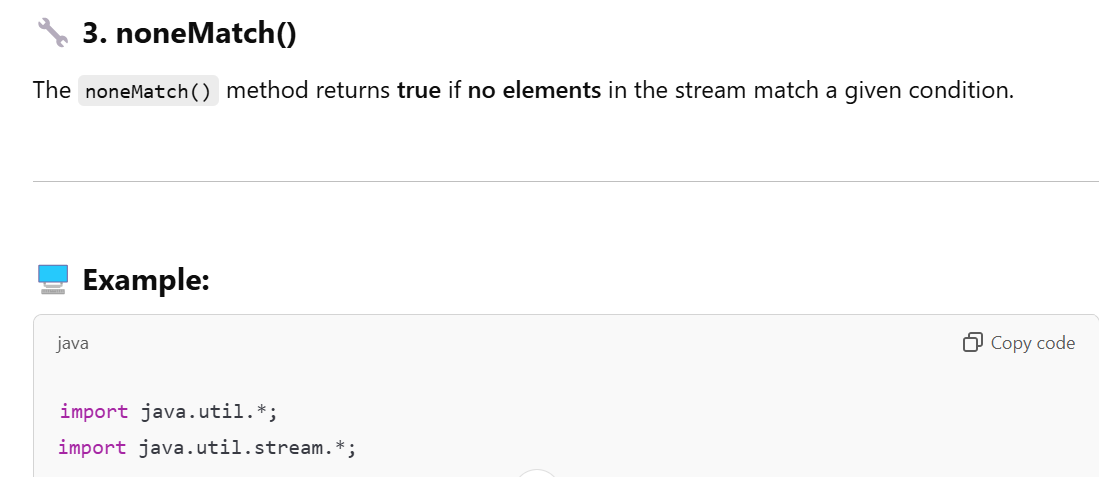
**🔧 1. anyMatch()**

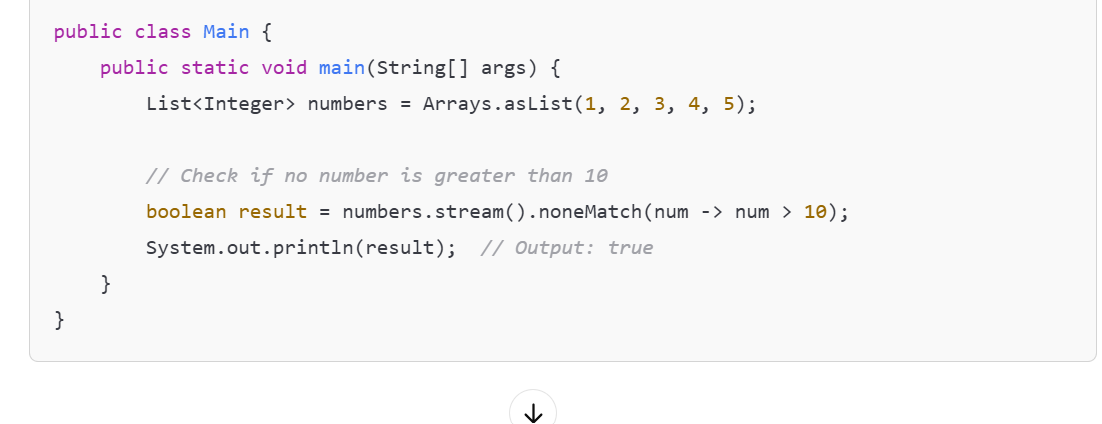
**The anyMatch() method returns true if any one element in the stream matches a given condition.**

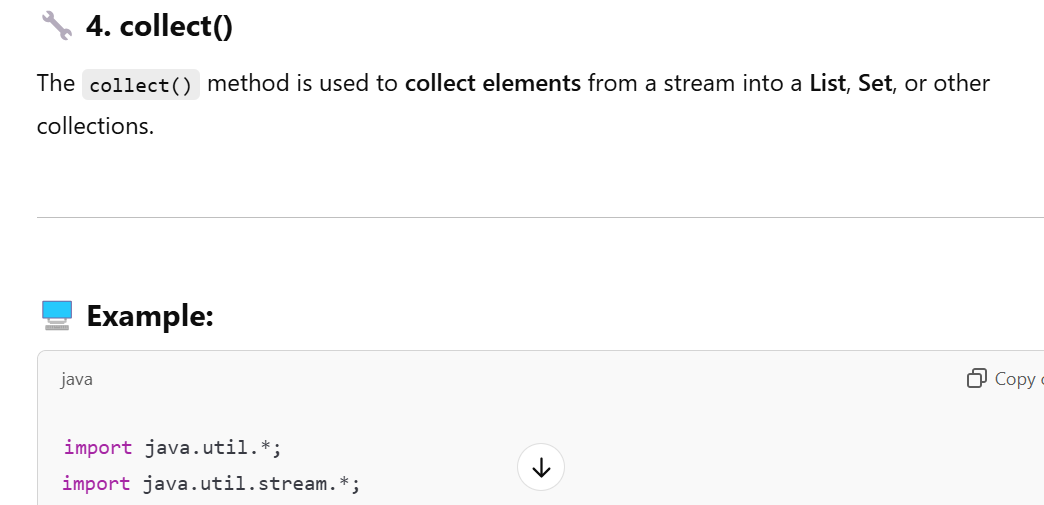
****

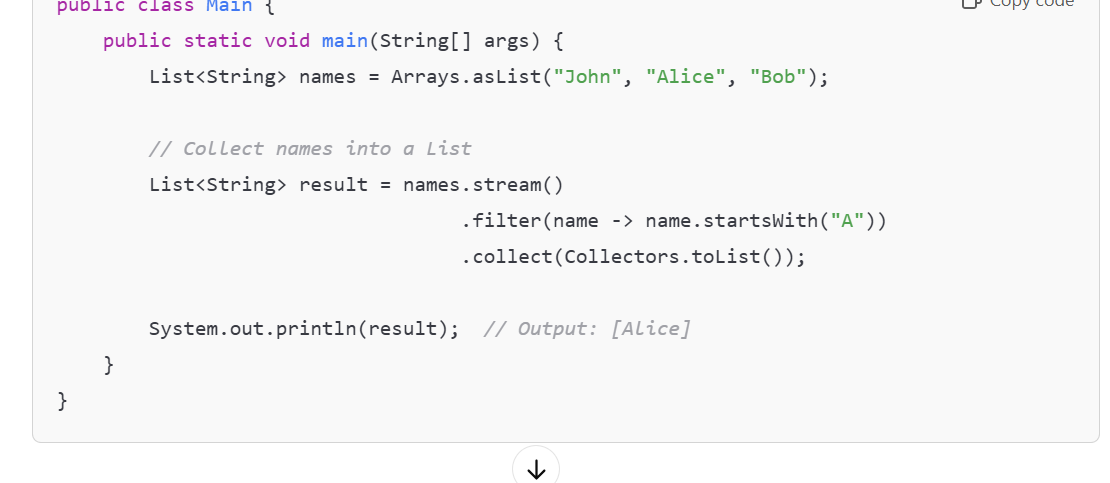
****

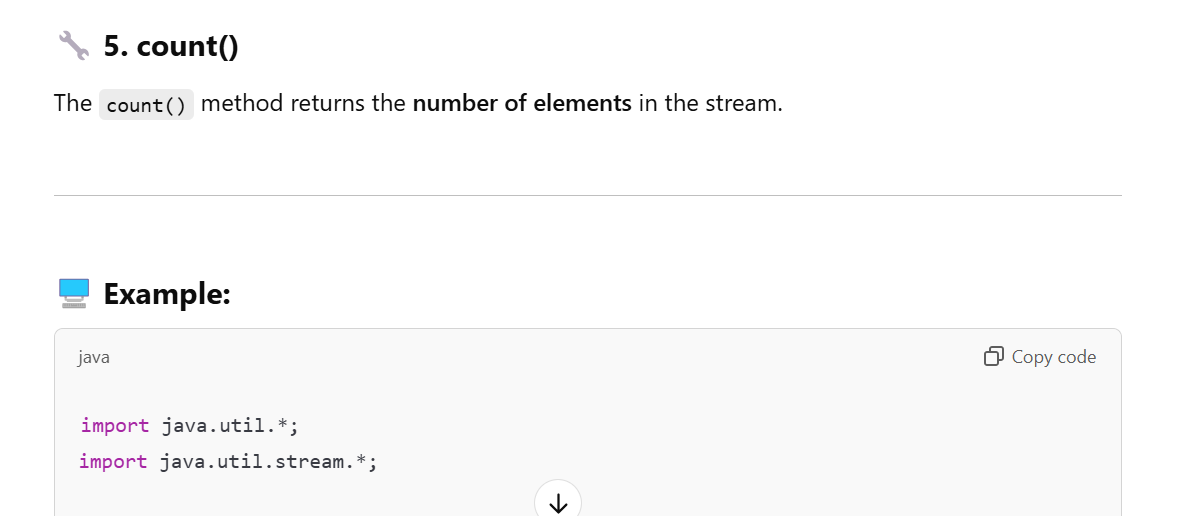
****

****

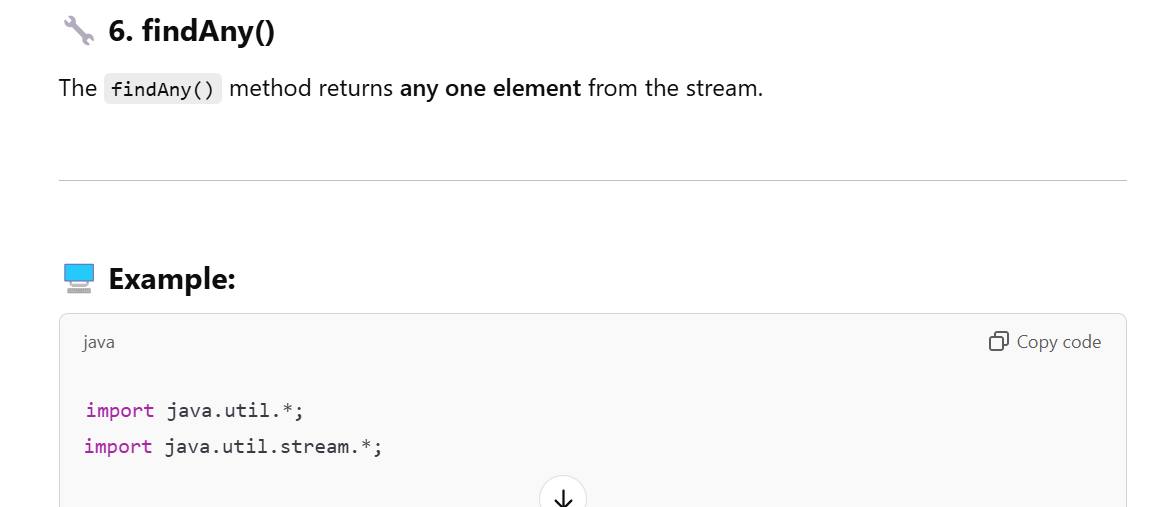
****

****

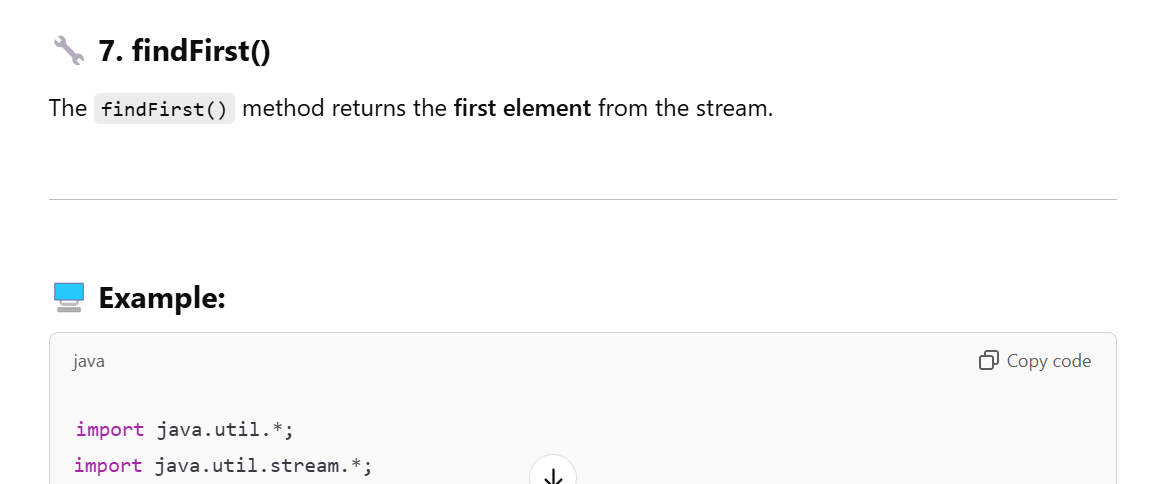
****

****

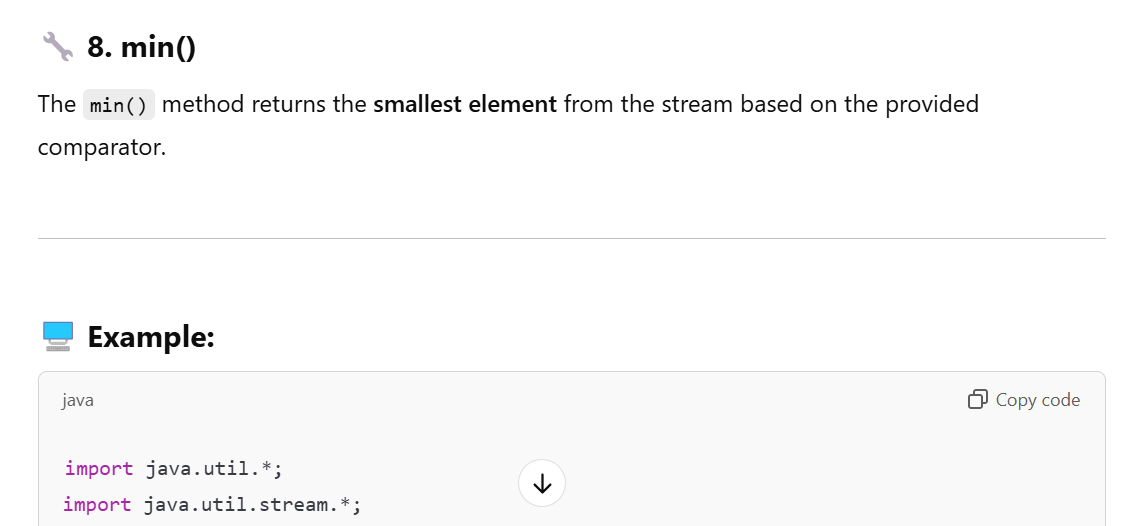
****

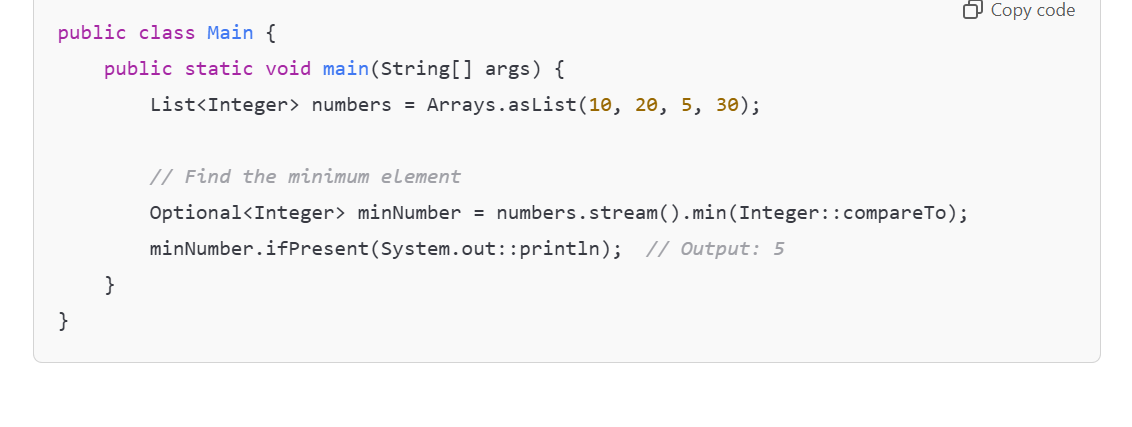
****

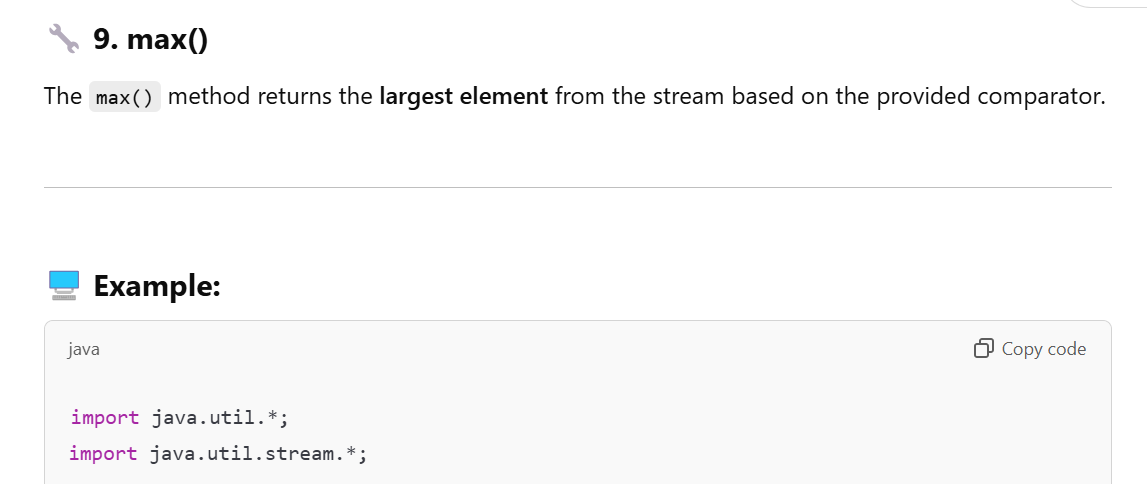
****

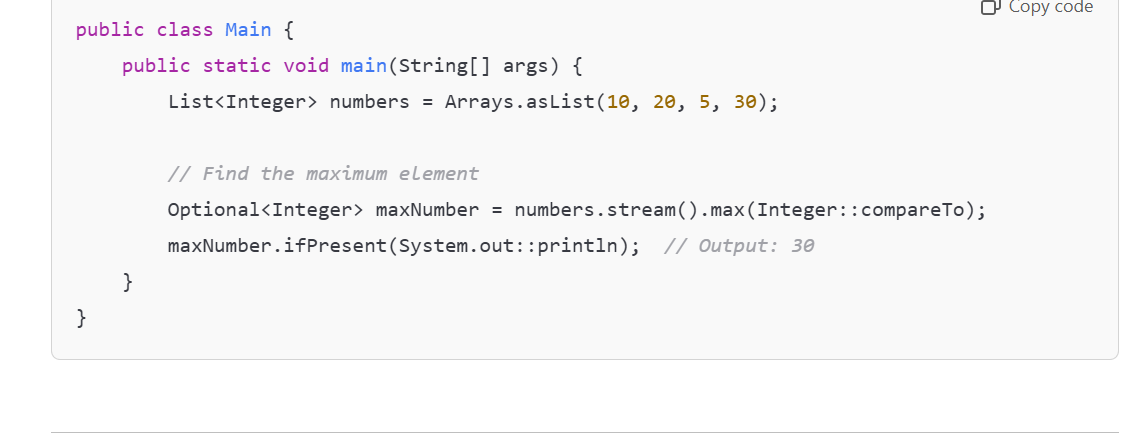
****

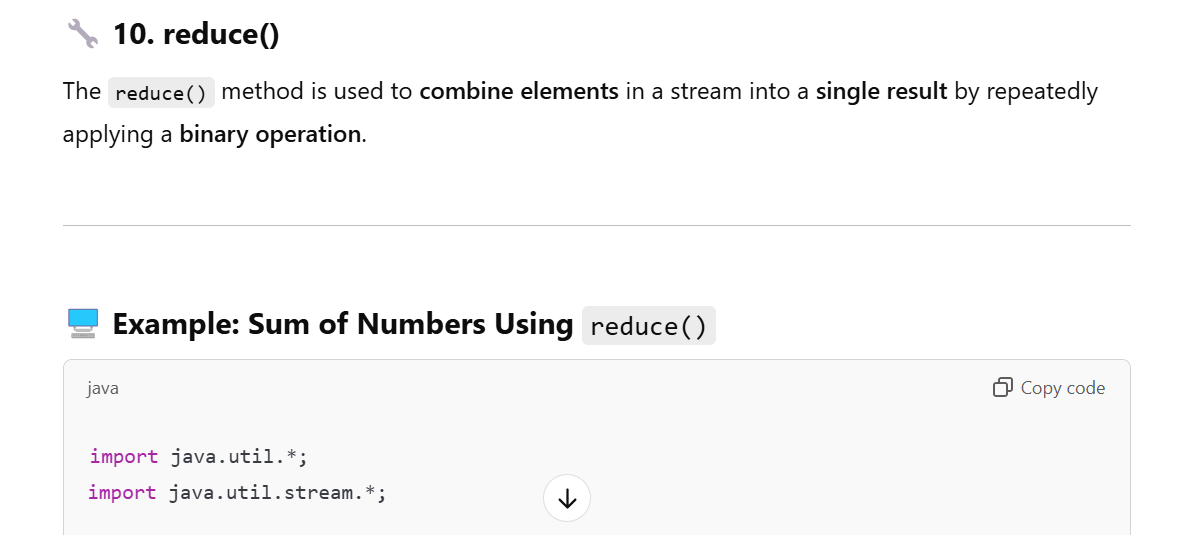
****

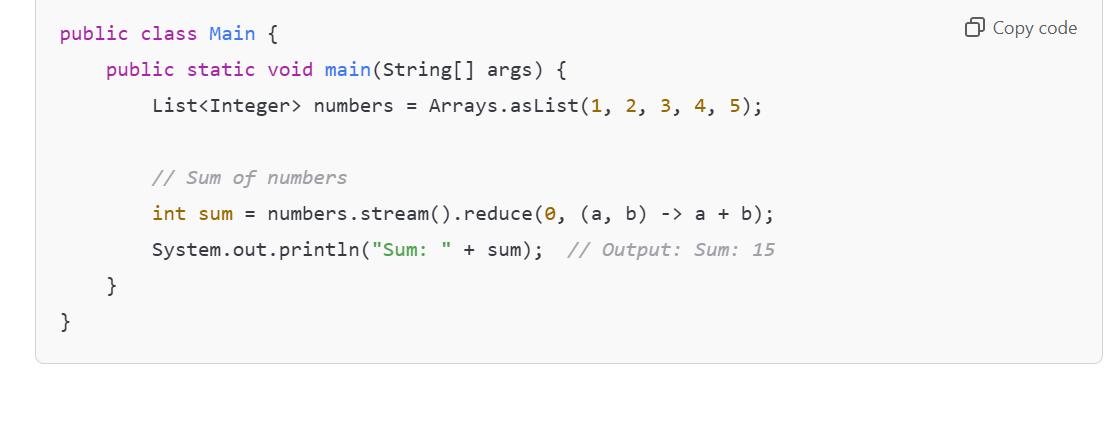
****

****

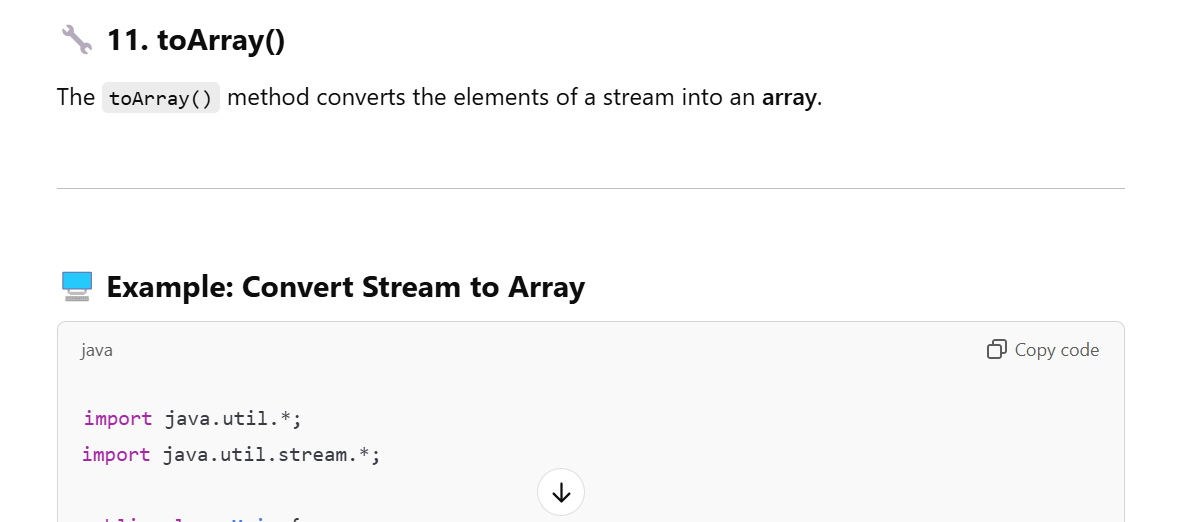
****

****

****

****

****

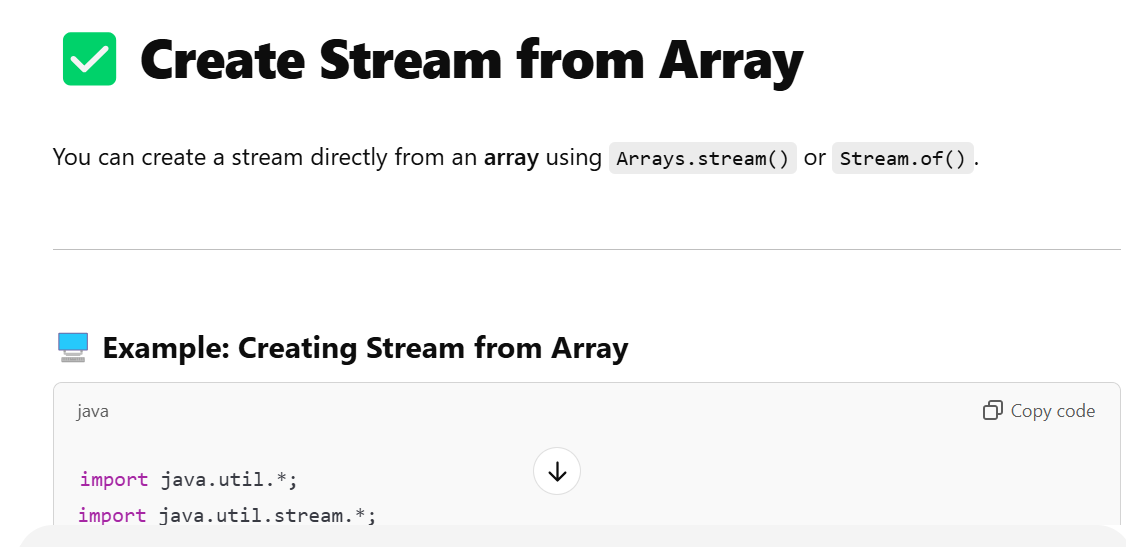
****

****

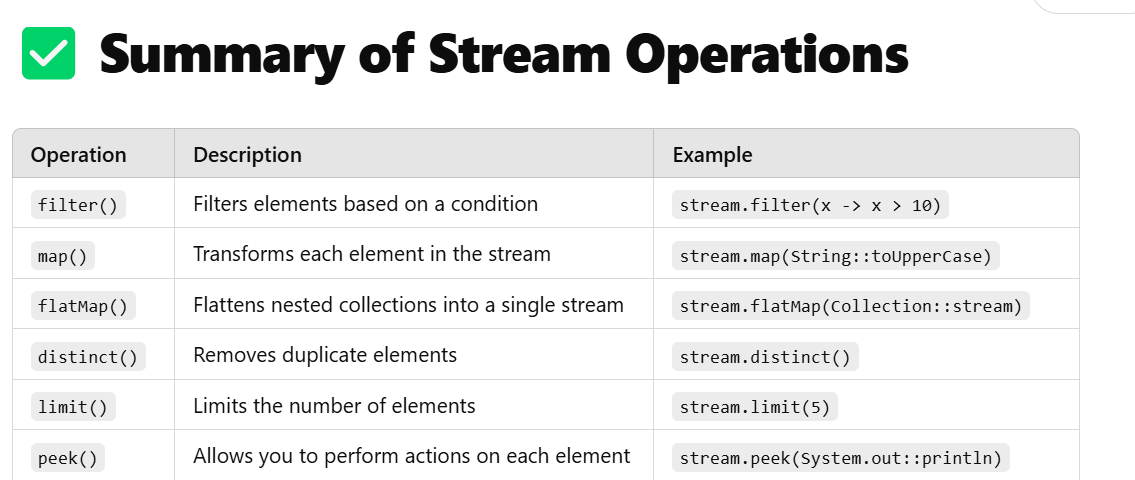
**✅ Concatenate Streams**

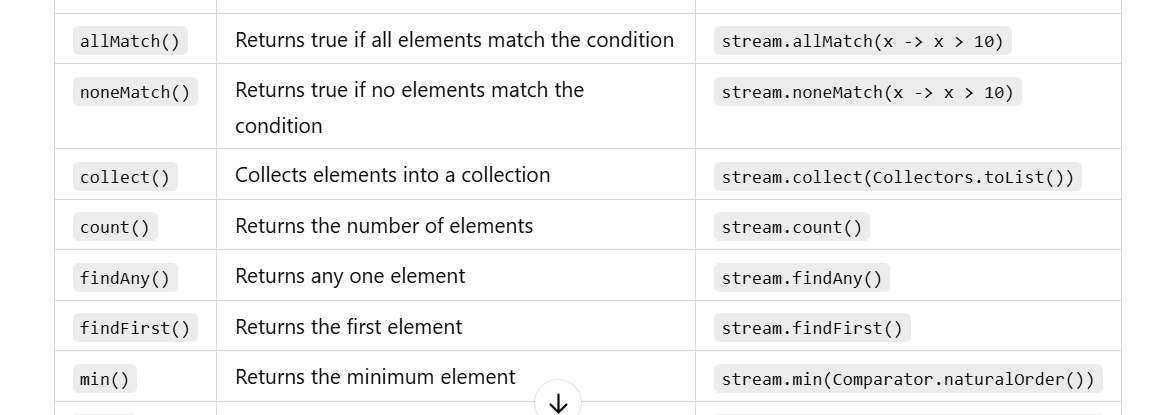
**You can concatenate two or more streams using the Stream.concat() method.**

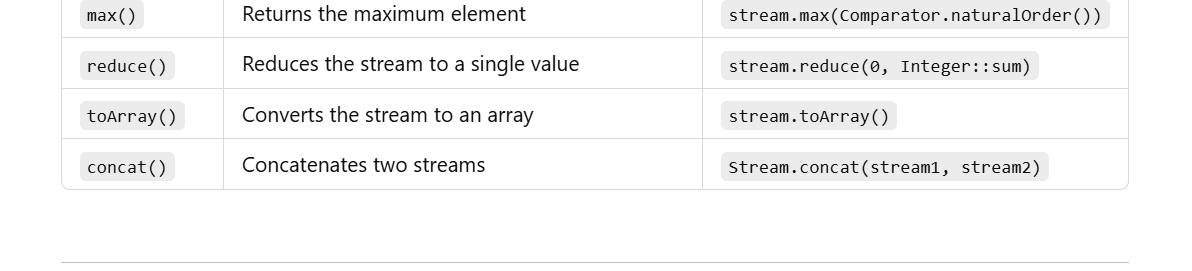
****

****

****

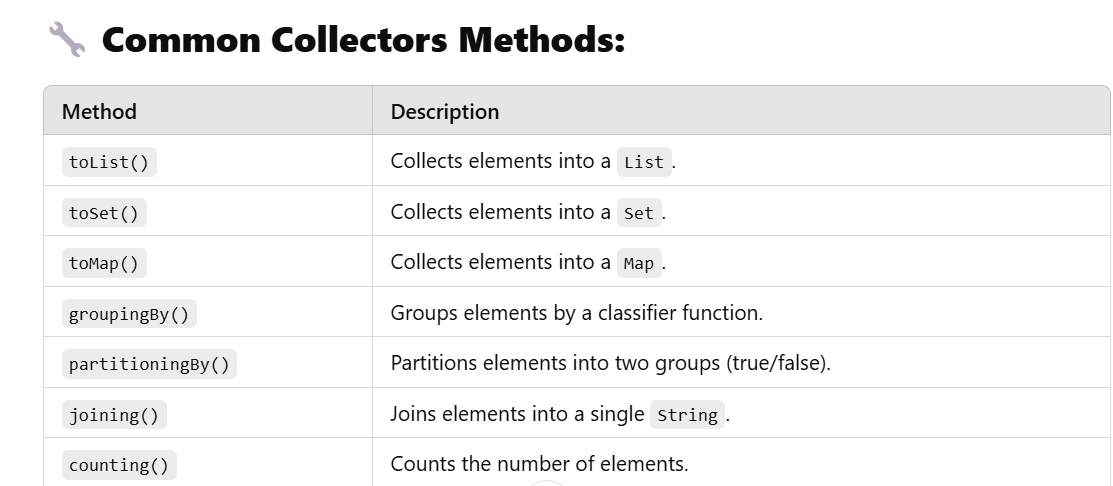
****

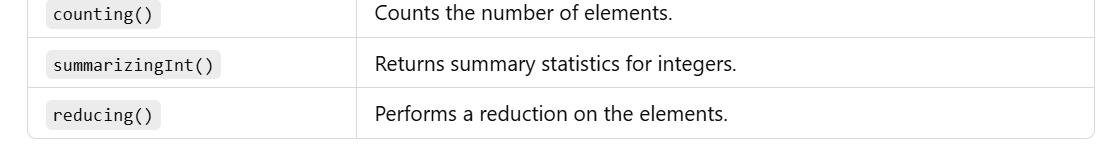
****

****

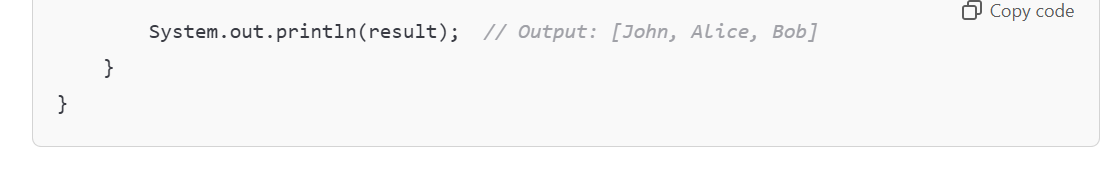
**✅ Collectors in Java Stream API**

**Collectors is a utility class in the java.util.stream package that provides various reduction operations such as converting streams to lists, sets, maps, and performing grouping, partitioning, and summarizing.**

****

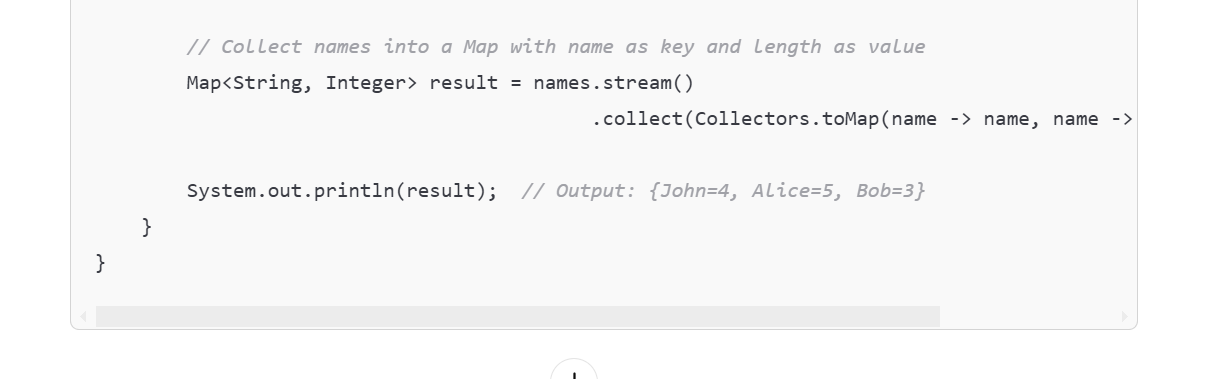
****

****

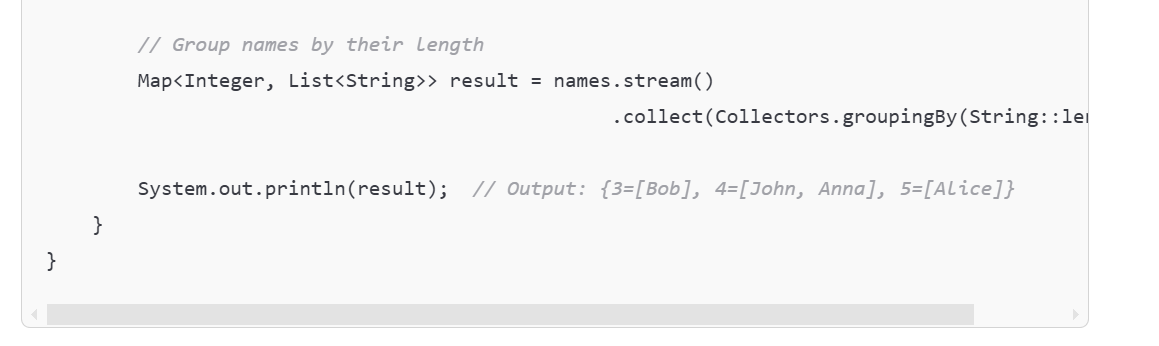
****

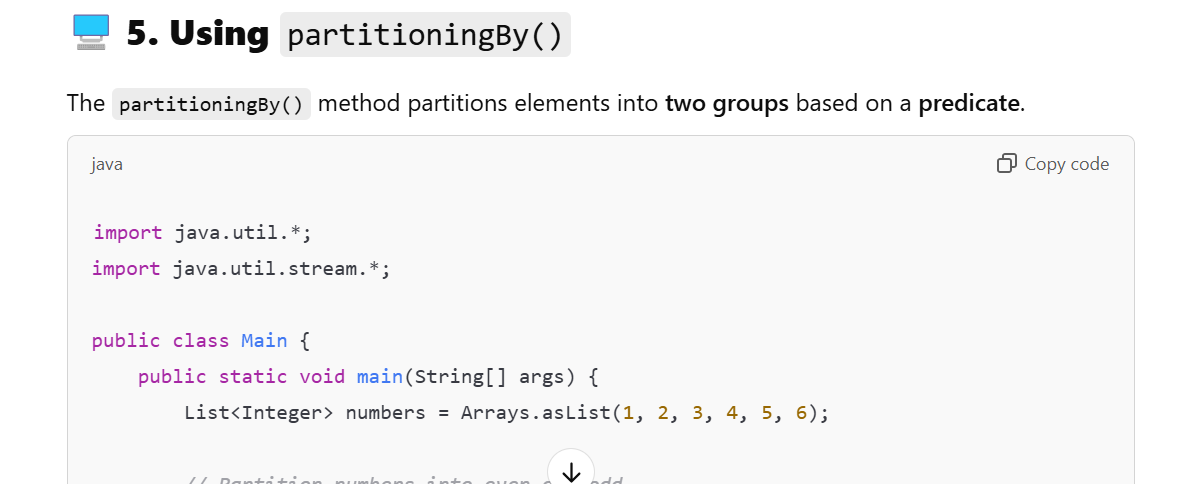
**** ****

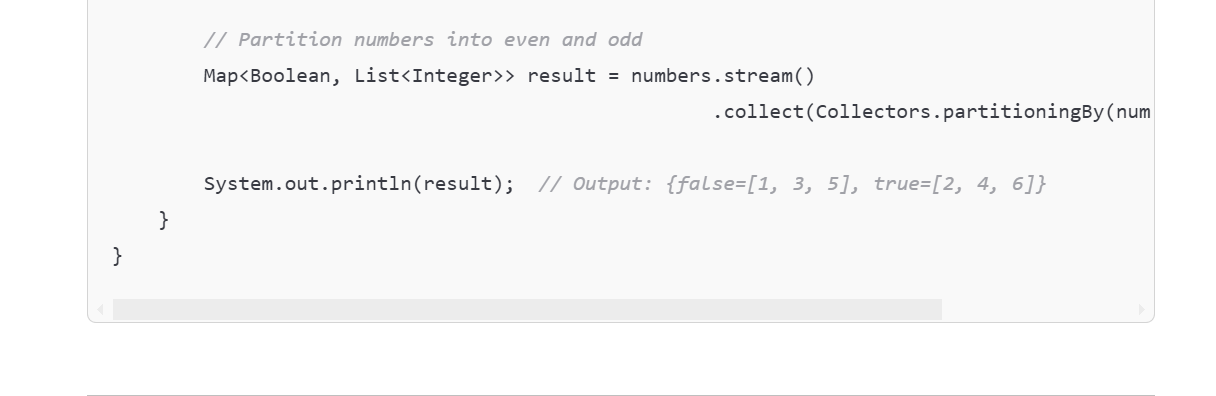
****

****

****

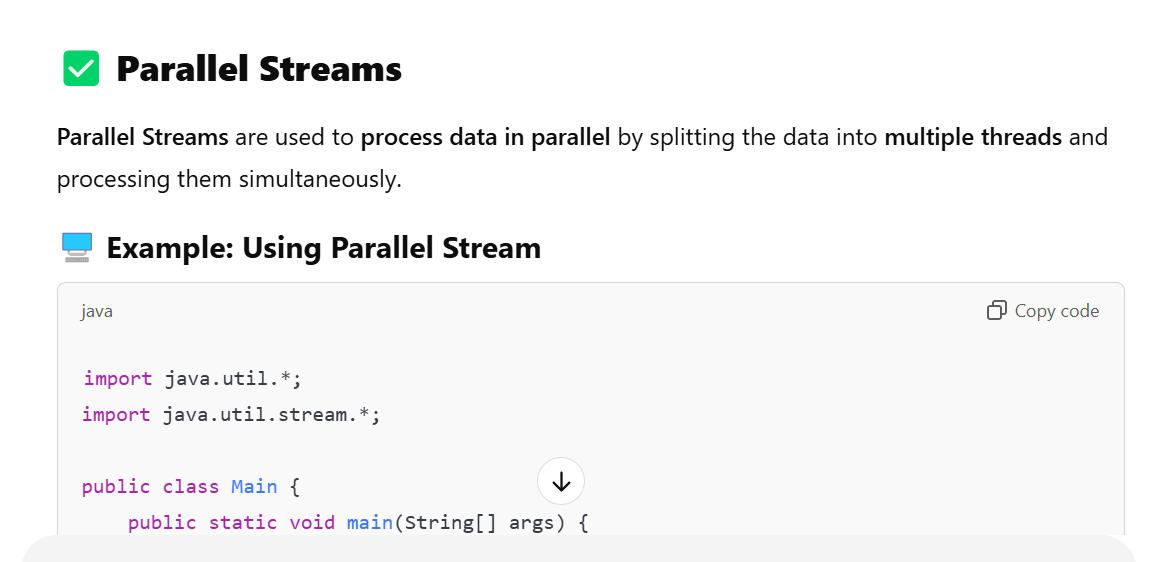
****

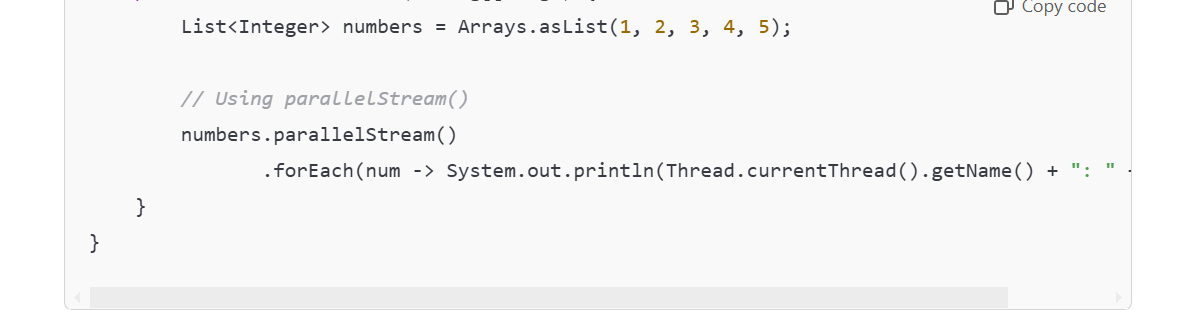
****

****

****

****

****

****

****

**🔧 When to Use Parallel Streams?**

* **Use Parallel Streams when you need to process large collections of data and want to reduce processing time.**
* **Do not use Parallel Streams when the task is I/O bound or involves shared mutable state.**